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

Christoph Pohl[1], Alf Zugenmaier[2], Michael Meier[3], and Hans-Joachim Hof[1](✉)

[1] MuSe - Munich IT-Security Research Group, Munich University of Applied Sciences, Munich, Germany
{christoph.pohl0,hof}@hm.edu
[2] Munich University of Applied Sciences, Munich, Germany
alf.zugenmaier@hm.edu
[3] Fraunhofer FKIE Cyber Defense, Bonn, Germany
michael.meier@fkie.fraunhofer.de

Abstract. Honeypots are used in IT Security to detect and gather information about ongoing intrusions by presenting an interactive system as attractive target to an attacker. They log all actions of an attacker for further analysis. The longer an attacker interacts with a honeypot, the more valuable information about the attack can be collected. Thus, it should be one of the main goals of a honeypot to stay unnoticed as long as possible. Also, a honeypot should appear to be a valuable target system to motivate attackers to attacks the honeypot. This paper presents a novel honeypot concept (B.Hive) that fulfills both requirements: it protects existing web application in productive use, hence offering an attractive attack target, and it uses a novel technique to conceal the honeypot components such that it is hard to detect the honeypot even by manual inspection. B.Hive does not need configuration or changes of existing web applications, it is web framework agnostic, and it only has a slight impact on the performance of the web application it protects. The evaluation shows that B.Hive can be used to protect the majority of the 10,000 most popular web sites (based on the Alexia Global Top 10,000 list), and that the honeypot cannot be identified by humans.

Keywords: Web application · Honeypot · Security · Web security · Network security

1 Introduction

Honeypots are well known and valuable components for the protection of networks. They can be used for attack detection or for research purposes. Usually, a honeypot is a *fake* system without any function that runs in parallel to other productive systems. Thus, all activities detected on the honeypot can be considered attacks (or unintended use). However, a honeypot can only monitor ongoing attacks if it succeeds in tricking attackers into attacking the honeypot at first. To do so, a honeypot must be known to an attacker and it should appear like a real

© IFIP International Federation for Information Processing 2015
H. Federrath and D. Gollmann (Eds.): SEC 2015, IFIP AICT 455, pp. 267–280, 2015.
DOI: 10.1007/978-3-319-18467-8_18

application or service. In order to maintain the attackers interest and to maximize the attackers interactions to gather as much information about the attack as possible, attackers should not be able to notice that the system they are attacking is a honeypot. The approach presented in this paper, B.Hive, blends into already existing and running web applications (further called target applications). As the honeypot components are completely invisible to benign users, any interaction with it is likely an attack. As an existing, productive web application is used, attacks on this system are likely.

The contribution of this paper is twofold: It presents a zero configuration Low Interaction Honeypot that can blend into any existing and running web application. Furthermore, it employs a technique that makes it substantially harder for an attacker to detect that honeypot components were integrated into a web application, even when manually inspected by humans. B.Hive does not need configuration for the integration, and the protected web application does not need to be changed. B.Hive is ideal to be integrated into active protection components like the web application firewall "All-Seeing Eye" [10].

The paper is structured as follows: The next Section 2 gives an overview on B.Hive. Related work is described in Section 3. Design and implementation is explained in Section 4. Section 5 validates the concept and shows that performance of the prototype implementation would allow augmentation of all but the busiest web applications. Section 6 summarizes the paper and gives an outlook on future work.

2 Overview

The Open Web Application Project (OWASP) maintains a list of the ten most prevalent attacks on web application in [7]. For four of these attacks, named A1 (Injection), A3 (Cross-Site Scripting (XSS)), A8 (Cross-Site Request Forgery (CSRF)), and A9 (Using Components with Known Vulnerabilities), an attacker usually inject malicious data into form fields of websites. As these attacks are very common, using form fields as a honeypot component allows a honeypot to detect many attackers and many different attacks. B.Hive transparently injects form fields into existing forms of the target application. To do so, B.Hive acts as a proxy between Internet and target web application. It intercepts web pages served by the target web application and modifies forms if present. Additional form fields are added to detected forms. Changes to these additional form fields are monitored to detect attackers inserting malicious data to test for common vulnerabilities (e.g. A1, A3, A8, A9, see above). As field manipulation is usually part of early phases of an attack (reconnaissance phase), detecting attacks at this point of time helps to monitor attacks. The fields injected by B.Hive can for example be hidden fields, or the fields are made invisible using CSS or JavaScript. In all cases, these fields are invisible to legitimate users of the web application. B.Hive also intercepts incoming HTTP requests to the target application and removes the injected fields again. Hence, B.Hive is invisible for the web application as well as legitimate users. There is no impact on the functionality of the web application.

The crucial point in injection fields into existing form field is to find suitable names and default values for the injected form fields. Most web applications use a consistent naming of form fields of a form, and the naming is consistent with the context of the web application. Hence, using random names as well as using the same name all the time is prohibitive. Involving web developers or administrators to define suitable field names for security components opposes the goal to build a zero configuration honeypot suitable for a large number of frameworks. It is the main contribution of this paper to propose a way to select suitable form field names and field parameters for the injected form fields. B.Hive selects suitable form fields and other parameters from a database of form fields harvested from a large number of existing applications. B.Hive detects the context of a form and selects a suitable field name and field parameters from this database.

3 Related Work

There are some approaches that use real applications to construct honeypots, for example [5]. However, the honeypot is directly integrated into the target application. Changing existing, already deployed applications is not desirable in a productive environment with already deployed applications. In contrast, B.Hive does not require changes of the target application. [3] describes an automated honeypot generation using search engines output. The resulting honeypot is a standalone non-productive web application. B.Hive in contrast protects an existing, productive web application.

Injection form fields in a form was already described in [9,12]. However, the developer has to implement these fields on his own in the target application or using jQuery. In both cases, the undetectability of the honeypot heavily depends on the developer to select suitable form field names and parameters. B.Hive does not need any configuration to adopt the look and feel from the original web application, hence relieves the developer from the burden of selecting suitable form field names and parameters. The approach in [6] also uses form fields as honeypot. In this case, form fields are duplicated and it is disguised, which is the form field to use. For a human, such a form is easy to spot. B.Hive in contrast puts special emphasize on staying undetected.

In comparison to related approaches, the presented zero configuration honeypot solution has the advantage that it integrates into the target application without the need of configuration. The integration is almost independent of target application technology, framework or system. The injected form fields adapt to the context of the web page in which they are injected to stay unnoticed even from manual inspection of the web page by a human attacker.

4 Design and Implementation

This Section describes the design of B.Hive with a special focus on the generation of suitable form fields for the forms to protect.

4.1 Generation of Plausible Fields

The goal of the form field generation is to generate a form field for injection that is plausible in the context of the form where it should be inserted. Plausible means that attackers as well as automated attack tools cannot distinguish inserted fields from original fields of the form. B.Hive tries to find plausible fields in a database of web forms harvested from the 10,000 most popular websites according to Alexa [1]. Important key figures of the Global Top 10,000 list of Alexa are described in Table 1.

Table 1. Initial database for Alexa Global Top 10,000

Websites	10,000
Extracted forms	15,255
Different field names	18,210
Average fields per form	3.8
Maximum fields per form	182
Minimum fields per form	0

For the purpose of optimized storage, the extracted form data gets preprocessed. In a first step, the attribute *name*, the field name (f), is extracted from every field. This attribute gets normalized as described in equation 1 where a character at index i in f is described as c_i. Ξ denotes a technical control character for further usage in B.Hive, Θ stands for an alphabet of lowercase letters, and Υ names an alphabet of uppercase letters. Allowed other characters are termed by ϑ. Let $u(x)$ be the function to bring an uppercase character to lowercase. The function $h(x)$ is used for preprocessing.

$$\text{For} \quad 0 \leq i < length(f)$$
$$h(c_i) = \begin{cases} c_i, & \text{if } c_i \in \Theta \vee c_i \in \vartheta \\ u(c_i), & \text{if } c_i \in \Upsilon \\ \Xi, & \text{if } (i-1 \neq 0 \wedge c_{i-1} \neq \Xi \wedge c_{i-1} \neq \emptyset) \\ & \quad \vee (i-1=0) \\ \emptyset, & \text{other} \end{cases} \tag{1}$$

Whenever $h(x) = \emptyset$, it will be ignored in further calculation. The result of this preprocessing gets stored in the new attribute f_{clean}.

The condition $b(x)$ to store f_{clean} in the B.Hive database is described in equation 2.

$$b(f_{clean}) = \begin{cases} true, & \text{if } length(f) > 0 \wedge f_{clean} \neq \{\Xi\} \\ false, & \text{other} \end{cases} \tag{2}$$

The final result f_{clean} is stored in a special Trie-structure (see [15] for details), optimized for calculation operations with the Levenshtein Distance [4]. The Levenshtein Distance is used in B.Hive as a metrics for similarity between field names. The Levenshtein Distance is defined as *"Minimum number of insertion, deletion and substitution edits required to change one word into another"* [4]. It is ideal to handle typical abbreviations used by web application developers, e.g. to compare field names like "passwd" and "password". Formally the Levenshtein Distance $lev(r, s)$ is defined in equation 3, using a search word s and a reference word r as input. The search word is compared to the reference word. The recursive function of $lev(r, s)$ is $k(|r|, |s|)$.

$$
\begin{aligned}
&\omega_1(i,j) = max(i,j) \\
&\omega_2(i,j) = min(k(i-1,j)+1, k(i,j-1)+1, \\
&\quad k(i-1,j-1)+0) \\
&\omega_3(i,j) = min(k(i-1,j)+1, k(i,j-1)+1, \\
&\quad k(i-1,j-1)+1) \\
&k(i,j) = \begin{cases} \omega_1(i,j) & \text{,if } min(i,j) = 0 \\ \omega_2(i,j) & \text{,if } min(i,j) \neq 0 \wedge r[i] = s[j] \\ \omega_3(i,j) & \text{,other} \end{cases}
\end{aligned}
\tag{3}
$$

The Trie-structure holds the preprocessed field names extracted from the Alexa Global Top 10,000 list. At the end of a field name (the last node in a Trie-structure) a link to the original field(s) is stored. Other parameters like field default values, raw HTML code, forms, and pages are stored in a separate database.

During run-time, the honeypot generator needs to find plausible form field names for the form fields that should be injected into forms in the output of the target application. Plausible means that attackers as well as automated attack tools cannot distinguish inserted fields from original fields of the form. This is done by finding forms in the candidate pool, which have similar form field names to the original response. B.Hive includes a LR-Parser with a state machine to extract form field names from the response. This means the full HTML source gets parsed. While parsing, it recognizes each form and each field of a form with its attributes. These forms and their form fields will be further used as input for B.Hive.

To find similar forms, it is necessary to define the similarity of field names (see equation 4). A field name is described with f and the length of f with $l(f)$. Φ is the set of all field names. Let $\Phi = \{f_1, f_2, \ldots, f_n\}$. A form F is described as $F \subseteq \Phi$ and the set of forms is denoted by Γ where $\Gamma = \{F_1, F_2, \ldots, F_n\}$. λ is a system parameter for tuning performance and precision. It describes the maximum acceptable Levenshtein Distance. The other system parameter δ ensures that short field names (shorter than $\delta + \lambda$) get compared with a lower Levenshtein Distance than longer field names. The key variable for the upper bound

of similarity (what is least similar) is denoted by μ.

$$\mu = \begin{cases} \lambda, & \text{if } min(l(f_1), l(f_2)) \geq \delta + \lambda \\ min(l(f_1), l(f_2) - \lambda, & \text{if } \delta \leq min(l(f_1), l(f_2)) < \delta + \lambda \\ 0, & \text{other} \end{cases}$$

For the similarity between two fields f_1, f_2 let

$$f_1 \sim f_2 \Leftrightarrow lev(f_1, f_2) \leq \mu \tag{4}$$

The calculation for the best matching form is described in equation 5. Based on the similarity between field names, the definition of similarity between two Forms F_1 and F_2 is: $F_1 \sim F_2 \Leftrightarrow \{\exists f_1 \in F_1 \exists f_2 \in F_2 : f_1 \sim f_2\}$. The function $a(F_1, F_2)$ describes the number of similar fields in F_1, F_2 where: $a(F_1, F_2) = |\{f_1 \in F_1 | \exists f_2 \in F_2 : f_1 \sim f_2\}|$. Further, the number of similar forms with a field similar to f is defined as $s(f)$ where: $s(f) = \sum_{F \in \Gamma} a(\{f\}, F)$. The set of different forms is denoted by Ψ (in contrast to Γ that could include similar forms). Ψ is defined as: $\Psi := \{F_1 \in \Gamma | \exists f_1 \in F_1 \forall F_2 \in \Gamma \setminus F_1 : \forall f_2 \in F_2 : lev(f_1, f_2) > 0\}$. To identify the best matching form Ψ_{Best} for a reference form R (the form of the target application that should be protected) equation 5 is used.

$$\Psi_{Best} = \{F_1 \in \Psi | \forall F_2 \in \Psi : a(R, F_1) \geq a(R, F_2)\} \tag{5}$$

In the last step, possible plausible fields for injection are identified. First, possible candidate fields Ω for injection are collected where:
$\Omega = \{f \in \bigcup_{F \in \Psi_{Best}} F | \forall r \in R : lev(r, f) > 0\}$.

Let $L^{[1]}$ be the list of field names of Ω descendingly ordered by the number of appearances in similar forms: $L^{[1]} = \{f_1, f_2, \ldots, f_n\}$. Such that: $i < j \Rightarrow s(f_i) \geq s(f_j))$.

Let $L^{[2]}$ be the list of field names of Ω descendingly ordered by the minimum Levenshtein Distance to any of the fields of the form in that the plausible field should be inserted: $L^{[2]} = \{f_1, f_2, \ldots, f_n\}$. Such that: $i < j \Rightarrow lev_{min}(f_i, R) \geq lev_{min}(f_j, R)$ where $lev_{min}(f, R) = \min_{r \in R}(lev(f, r)$ The index of f in L^k is denoted by $index_k(f)$.

The result score $score(f)$ for a field f is defined by: $score(f) = (\alpha * index_1(f)) + (\beta * index_2(f))$ where α, β are factors to weight the ordering of $L^{[1]}$ and $L^{[2]}$. In this approach, let $\alpha = \beta = 1$ List $L^{[3]}$ is the list of the field names of Ω ascendingly sorted by the result score $score(f)$. $L^{[3]} = \{f_1, f_2, \ldots, f_n\}$. Such that: $i < j \Rightarrow score(f_1) \leq score(f_2)$.

The field with the lowest $score(f)$, respectively the first field in $L^{[3]}$, is selected by B.Hive as the most plausible field name.

4.2 Position of Form Fields

For the injected field to be unnoticed, it is necessary to find a plausible position of the injected form field in the form. B.Hive will inject the honeypot field at a

position in the target form F than is similar to the position in the form H from that the honeypot field was harvested.

$L^{[H]}$ is a list of field names from form H ascendingly ordered by the index of the field names in H: $L^{[H]} = (h_1, h_2, \ldots, h_i)$. $L^{[F]}$ defines a similar list for the form F: $L^{[F]} = (f_1, f_2, \ldots, f_i)$. Let h_k be the honeypot field to inject into the target form F. Let l be the index, where $|l - k|$ is minimal and $\exists m : f_m \sim h_l$. The result form is defined in equation 6:

$$L^{[F']} = \begin{cases} (f_1, f_2, \ldots, f_{m-1}, h_k, f_m, \ldots, f_j) & \text{,if } l - k \geq 0 \\ (f_1, f_2, \ldots, f_m, h_k, f_{m+1}, \ldots, f_j) & \text{,if } l - k < 0 \end{cases} \tag{6}$$

4.3 Field Type and Default Value

In most of the cases from the Alexa Top 10,000 the type of a fields with the same name is the same. Hence, it is possible to let the injected field have the same type and default value as any one of the fields in the database. B.Hive injects the honeypot field using the same type and default value it had in the form from which it was originally harvested. The fields are hidden by hidden attribute. Whenever the field from the result form contains an id, the honeypot field will get this id too, except this id already occurs in the original page. The algorithm for the ordering of the attributes is naive but effective. B.Hive computes the most frequently used ordering from the original page. As ordering attributes, *name, value,id* and *style* is used. The injected form field gets constructed with this ordering.

5 Evaluation

This chapter provides the evaluation results of B.Hive. First, the choice of system parameters for the evaluation is presented. Subsection 5.2 evaluates the effectiveness of B.Hive. The following subsection evaluates the quality of the honeypot. The last subsection evaluates the performance of B.Hive.

Every analysis uses the full set of data without snipping outliers. For the sake of readability, histograms only show forms with less then 16 fields. Only 292 out of the 15,255 harvested forms have more than 15 fields.

5.1 Choice of System Parameters

The most relevant system parameter for the performance and the effectiveness of the honeypot is the maximum edit distance λ. When choosing λ there are two computing factors: Whenever the allowed Levenshtein Distance grows, the similarity check gets more accurate but the performance drops.

For the evaluation, one honeypot field for every Alexa Top 10,000 has been generated with different values for λ in the range $\{0, 1, \ldots, 5\}$. Table 2 shows

Table 2. Percentage of cases in which no plausible field could be found and run-time for different values of λ

λ	**No result**	**∅ run-time sec**
0	9.34%	0.026
1	6.90%	0.079
2	5.20%	0.227
3	**3.78%**	**0.459**
4	2.64%	0.731
5	1.99%	1.027

the resulting run-time as well as the percentage of cases where no plausible field could be found for different values of λ. B.Hive has been started single threaded with a sequential calculation.

A value of $\lambda = 3$ was chosen for all other evaluations as it provides a balanced result for run-time and success rate. The system parameter δ was set to $\delta = 3$. Changing this parameter to a lower variable has no significant changes in the accuracy, but the subjective quality of honeypot fields drops in some cases. The subjective quality has been measured with a manual validation of the results.

5.2 Evaluation of Effectiveness

To prove that it is possible to generate honeypot fields for most existing web application, B.Hive was used to generate honeypot fields for each website of the Alexa Top 10,000 (list of most popular websites worldwide).

Table 3. Results of the Evaluation of Effectiveness of B.Hive when protecting each website of the Alexa Top 10,000 list

Number of forms	15,255
Trie-Nodes	140,298
Field names	18,210
Protectable forms	146,790 ∼96.22%
∅ similar fields	2.5
∅ possible honeypot fields / form	1,023.4

Table 3 shows the results: A significant number of forms (96.22 % of all forms) can be protected by B.Hive. Successful protection of a form means in this context, that at least one plausible field was found for the form. B.Hive keeps a list of unprotectable forms. Whenever there is no plausible field for a form (3.78 % of all forms), B.Hive takes a random field from the list of unprotectable forms. In the following evaluation, this is not regarded as success.

The evaluation of the effectiveness shows, that in average there are 2.5 similar fields in the target form and the form from which a honeypot field is taken. In

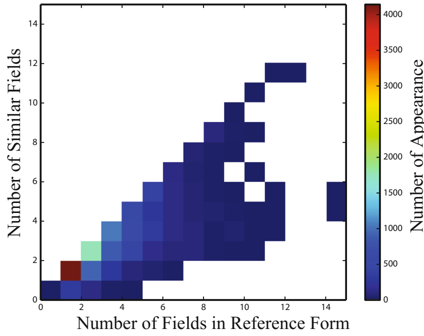

Fig. 1. Field similarity metric

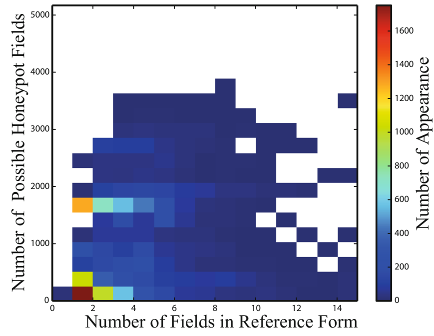

Fig. 2. Possible honeypot fields/form

average, for each form there are 1023.4 possible honeypot fields. Figure 1 and Figure 2 describes these results in detail.

Figure 1 gives a more detailed view on the similarity of fields. This figure shows the similarity between the target form and the result form for each form in a site of the Alexa Top 10,000 list. This and all further histograms use less equal calculation. This means that the upper right corner of a bin represents the value. Figure 1 has the number of fields in the reference form (target form) on the x-axis. On the y-axis the number of fields in the similar form (the form from which the honeypot field was chosen by B.Hive) is given. The bins and their colors represent the number of occurrences of a combination.

Figure 2 shows the number of potential honeypot fields per target form for all forms of the Alexa Top 10,000 list. The x-axis describes the number of fields in the target form. The y-axis shows the number of potential honeypot fields for this form. The color denotes the number of occurrences of this combination. It could be seen that there is a significant number of potential honeypot fields for all but very small forms (forms with only one field).

In conclusion B.Hive is able to generate a proper protection for the vast majority of the most popular 10,000 web sites.

5.3 Evaluation of Honeypot Quality

It is one of the main goals of the work presented in this paper to keep an attacker unaware of the presence of the honeypot. Hence, it is very important, that humans (attackers) cannot identify the injected form fields when manually inspecting the HTML source code of a page. To evaluate, if attackers can identify injected form fields, an empirical study with 75 participants was executed. All participants are students in a computer science program (49 in a bachelor program, 26 in a master program). Participants of the study were presented the content of 50 different forms of the Alexa Top 10,000 list. With a probability of 0.5, a form field was injected by B.Hive, in the other cases the original form was presented. The students were told before that 50% of all forms include an injected

Table 4. Empirical study setting

Students	75
Evaluated number of forms	3,714
Forms without honeypot	1,798 (48.41%)
Forms with honeypot	1,916 (51.59%)

field. The students were asked to identify injected form fields or to state that there is no injected field in a form. The default answer for each form was *"I do not want to answer"* to make sure, that no "click-through" influences the results. Table 4 summarizes the setting. The students decided in 1,919 cases (51.67%) that a form includes an injected field. In 1,013 cases (52.79% or 27.28% of all choices) they were right and in 906 cases(47.21% or 24.39% of all choices) they were wrong. In 1,675 cases (45.1%) the students decided that no injected field was present. In 825 cases (49.25% or 22.21% of all choices) they were correct and in 850 cases (50.75% or 22.89% of all choices) they were wrong. In the remaining 120 cases (3.23% of all choices) the students did not want to answer.

In average, the students chose in 24.745 (out of 50) cases (49.49%) the correct answer (field injected or not). The standard deviation for a correct answer is 6.74 with a maximum of 35 correct answers and a minimum of 1 correct answer.

These results are significantly near random choices, hence showing that it is not possible for an attacker to identify the injected form fields when looking at the field names.

In 314 cases the injected field was detected. In average the students decided in 4.077 (out of 50) cases (8.154%) for the injected field and in median they decided in 3 cases for the injected field. The standard deviation is 2.818 with a maximum of 12 correct answers and a minimum of 0 correct answers for all student and 50 answers. With random choices the probability to detect the honeypot field is 10.748% (with 4.8 fields per form in average when a field was injected).

This result is also significantly near random choices.

There has been no significant difference between master and bachelor students.

In conclusion, the evaluation shows that B.Hive is able to hide itself in the vast majority of forms. Humans cannot successfully identify the injected form fields.

In order to show that B.Hive is useful for detecting automated attacks, and that attack tools do not avoid the fields injected by B.Hive, the breakable web application (BREW) [11] was augmented with B.Hive and then attacked using penetration testing tools Owasp Zed Attack Proxy Project (Owasp ZAP) [8] and Vega [13].

In average, each form of BREW has about 2.43 fields. In conclusion it is expected that a penetration testing tool will hit the honeypot field with about 40% of all requests. Table 5 concludes the result in one overview. In all cases B.Hive worked correct and the penetration testing tools identified all B.Hive fields as possible target. The row *touch quota* describes the expected calls to the

Table 5. Validation with penetration testing tools

	Owasp ZAP	**Vega**	$\sum$
Requests	783	2,097	2,880
Post Requests	524	814	1,338
Trapped Requests	206	342	548
Touch quota	99%	101%	100%

honeypot when the penetrations testing tool identify the honeypot as suitable target.

In conclusion, each penetration testing tool recognized the injected honeypot field as a possible target. Both tools showed the expected amount of attacks on the target. B.Hive was able to recognize and identify each attack.

5.4 Performance Evaluation

B.Hive works as a proxy for web applications, so all traffic to the target application passes B.Hive. Hence, it is important to evaluate if B.Hive is ready for productive usage.

Figure 3, 4 and 5 show the performance of B.Hive **without** caching and **without** the overhead of parsing.

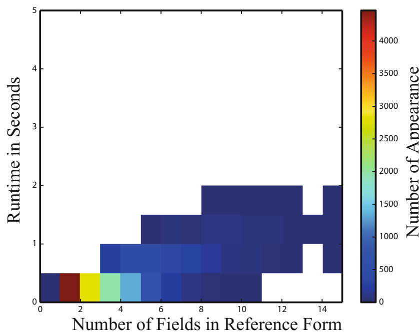

Fig. 3. Run-time of B.Hive without caching and without overhead of parsing

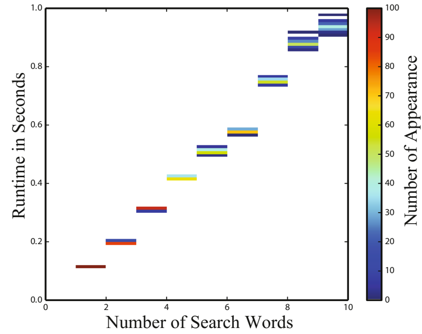

Fig. 4. Performance number of search words

In contrast, Figure 6 shows B.Hive under productive usage **with** enabled caching and **with** parsing.

Figure 3 shows the run-time for B.Hive for forms of the Alexa Top 10,000 list with a different number of fields in them. The x-axis shows the number of fields in the target form. The y-axis shows the run-time in seconds to find a similar field. The colored bins describes the number of appearance for this combination. It can be seen that B.Hive is able to protect a new website with a

proper run-time. The vast majority of forms can be protected under 0.5 seconds. The average run-time for B.Hive is 0.46 seconds. However, for productive usage, it is highly recommended to use caching for optimized run-time.

Figure 4 shows the run-time for B.Hive with different number of form fields in the target form. The search words used are randomized strings with a length of ten characters. It is guaranteed that the field names of the target form have no similarity to any other word in the database. This avoids side effects during result set building. Every number of field names has been measured 100 times. The x-axis describes the number of reference words per evaluation. The y-axis shows the run-time of B.Hive. The color denotes the number of occurrences of this combination.

The evaluation shows that the run-time grows near linear with the number of form fields in the target form, hence protecting forms with a low number of fields is faster then protecting forms with a high number of field. Fortunately, the evaluation of the Alexa Top 10,000 showed, that the average number of fields per form is very low (average of 3.8).

Figure 5 shows the dependency between the run-time of B.Hive and the length of one field name. The x-axis shows the length a of field name. The y-axis

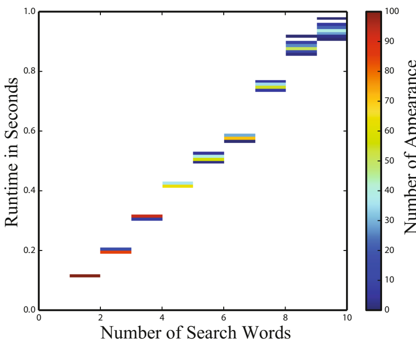

Fig. 5. Performance diff. word length

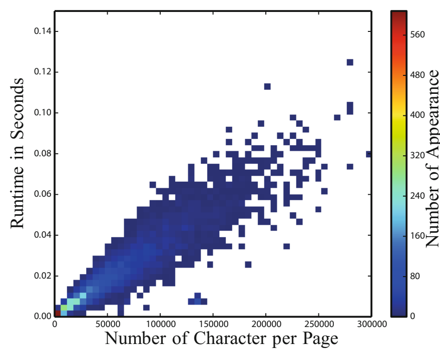

Fig. 6. Performance with caching and with parsing

shows the measured run-time. Each length between $1 - 50$ has been measured for 100 times. The figure shows that the dependency is near linear. The falloff between a length of 1 and 6 shows the effect of the system parameter δ.

The B.Hive algorithm is designed with the possibility of multi-threading. Each form field name analysis is atomic, a number of form field names (forms with more fields) can be calculated in a parallel way. The result set assembling is designed that algorithm like map reduce [2] can be used.

In conclusion, the performance evaluation shows that without caching (or the protection of a new form, not known to the cache) B.Hive is able to protect a web application in productive usage.

Figure 6 shows B.Hive in a productive scenario: caching and parsing is enabled. Injected form fields get cached and further injections do not have to run the algorithm again but can look up a suitable field in the cache. The x-axis of the figure shows the number of characters for each page. The y-axis shows the measured run-time for each page. The colored bins shows the number of appearances of each combination. The figure shows the correlation between the number of characters in the raw HTML page and the runtime, which is near linear. In average, B.Hive needs 30.5258 milliseconds to protect one page, which is 15.072 times faster than without caching. The number of forms per page does not correlate with the runtime. The overhead to query the cache (measured without parsing) is insignificant with 0.000072 milliseconds in average.

The honeypot field generation for one page is done by a single process. A load balancing with more processes or different server can be done by starting more instances of B.Hive and a load balancer like nginx [14].

In conclusion, B.Hive is able to protect even large and busy web applications when using caching and parsing.

6 Conclusion and Outlook

This paper presents B.Hive, a honeypot that protects existing web applications in productive use by transparently adding form fields to forms with a special focus on the undetectability of the honeypot by human inspection. The evaluation of B.Hive shows that humans are not able to identify the injected form fields, hence an attacker cannot avoid the honeypot. This allows to gain valuable insights into attacks. The evaluation also showed, that B.Hive only adds a slight overhead to the total response time of a web application when using caching and parsing. It also shows that B.Hive can protect the vast majority of web applications.

Over the course of the next year we plan to deploy B.Hive on a public web server to gather data on how real attackers interact with it. This could also lead to classification of attack payloads. Future work includes a extension of B.Hive beyond form field injection.

References

1. Alexa Internet, I.: Alexa - The Web Information Company. http://www.alexa.com/ (last accessed March 13, 2014)
2. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)
3. John, J.P., Yu, F., Xie, Y., Krishnamurthy, A., Abadi, M.: Heat-seeking honeypots. In: The 20th International Conference, p. 207. ACM Press, New York (2011)
4. Levenshtein, V.I.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Soviet Physics Doklady **10**, 707 (1966)
5. Mueter, M., Freiling, F., Holz, T., Matthews, J.: A generic toolkit for converting web applications into high-interaction honeypots. University of Mannheim (2008)

6. Nassar, N., Miller, G.: Method for two dimensional honeypot in a web application. In: 2012 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), pp. 681–686 (2012)
7. OWASP: Top 10 2013 - OWASP. https://www.owasp.org/index.php/Top_10_2013 (last accessed March 13, 2014)
8. Owasp: OWASP Zed Attack Proxy Project - OWASP (2014). https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project (last accessed October 23, 2014)
9. Perry, K.: Honeypot Technique of Blocking Spam - Dex Media, May 2013. http://www.dexmedia.com/blog/honeypot-technique/ (last accessed October 20, 2014)
10. Pohl, C., Hof, H.J.: The all-seeing eye: a massive multi-sensor zero-configuration intrusion detection system for web applications. In: SECURWARE 2013, The Seventh International Conference on Emerging Security Information, Systems and Technologies (2013)
11. Pohl, C., Schlierkamp, K., Hof, H.J.: BREW: a breakable web application. In: European Conference of Software Engineering Education, ECSEE 2014, November 2014
12. Squiid: Honeypot: Protecting web forms * Squiid, June 2011. http://squiid.tumblr.com/post/6176439747/honeypot-protecting-web-forms (last accessed October 20, 2014)
13. SubGraph: Vega Vulnerability Scanner (2014). https://subgraph.com/vega/ (last accessed October 23, 2014)
14. Sysoev, I.: nginx (2014). http://nginx.org/ (last accessed October 23, 2014)
15. Wang, Y., Peng, T., Zuo, W., Li, R.: Automatic filling forms of deep web entries based on ontology. In: Web Information Systems and Mining, pp. 376–380 (2009)