Proceedings of the 53rd Hawaii International Conference on System Sciences | 2020

HoneyBug: Personalized Cyber Deception for Web Applications

Amirreza Niakanlahiji
Univ. of Illinois Springfield
aniak2@uis.edu

Jafar Haadi Jafarian
Univ. of Colorado Denver
haadi.jafarian@ucdenver.edu

Bei-Tseng Chu
UNC Charlotte
billchu@uncc.edu

Ehab Al-Shaer
UNC Charlotte
ealshaer@uncc.edu

Abstract

Cyber deception is used to reverse cyber warfare asymmetry by diverting adversaries to false targets in order to avoid their attacks, consume their resources, and potentially learn new attack tactics. In practice, effective cyber deception systems must be both attractive*, to offer temptation for engagement, and* believable*, to convince unknown attackers to stay on the course. However, developing such a system is a highly challenging task because attackers have different expectations, expertise levels, and objectives. This makes a deception system with a static configuration only suitable for a specific type of attackers. In order to attract diverse types of attackers and prolong their engagement, we need to dynamically characterize every individual attacker's interactions with the deception system to learn their sophistication level and objectives and personalize the deception system to match with their profile and interest. In this paper, we present an adaptive deception system, called* HoneyBug*, that dynamically creates a personalized deception plan for web applications to match the attacker's expectation, which is learned by analyzing their behavior over time. Each HoneyBug plan exhibits fake vulnerabilities specifically selected based on the learned attacker's profile. Through evaluation, we show that HoneyBug characterization model can accurately characterize the attacker profile after observing only a few interactions and adapt its cyber deception plan accordingly. The HoneyBug characterization is built on top of a novel and generic evidential reasoning framework for attacker profiling, which is one of the focal contributions of this work.*

1. Introduction

In cyberspace, the relationship between attackers and defenders is highly asymmetric. On the one hand, attackers can discover vulnerabilities and exploit the target systems in a highly stealthy way. On the other hand, defenders are usually unaware of the attackers' techniques or skills. This asymmetry makes cyber defense much harder than cyber offense. Cyber deception systems offer new capabilities to reverse this asymmetry by (1) learning important information about attacker's capabilities and goals, (2) diverting attackers to false targets and wasting their resources (e.g., time and effort), and (3) slowing down the attackers in such a way that allows the defender to timely respond and harden the security.

Traditional honeypot systems such as honeyd [1] or nepenthes [2] are examples of cyber deception systems that construct fake computing systems resembling real ones for the sake of attracting attackers. If an attacker interacts with a honeypot, this will allow for observing the attacker's actions and learning their attack strategies and goals, while keeping the attacker distracted from reaching the real assets.

A major drawback of most existing deception systems is their static nature that makes them incapable of blending with the surrounding environment and reshaping their structures based on the attackers' characteristics. Blending with the environment (*e.g.*, by mimicking surrounding cybersystem configurations) is important to reduce the attackers' suspicion. However, it is not enough, and a deception system should also have the ability to reshape (adaptive reconfiguration) itself to engage the attacker for a longer time.

Most of the existing honeypot systems have static configurations. In such systems, administrators have to decide about the types and sophistication levels of vulnerabilities the system must expose. Adding too many vulnerabilities can make the system less believable (untrusted) by attackers. On the other hand, enabling too few vulnerabilities can make the system less compromisable; thus, making it less attractive.

Several works have presented the idea of context-aware deception systems such as Dynamic Honeypot [3] and HIDE which can adaptively blend with the environment. However, the adaptation is only based on the changes in configurations of neighbor

URI: https://hdl.handle.net/10125/63972
978-0-9981331-3-3
(CC BY-NC-ND 4.0)

systems. To the best of our knowledge, none of the existing works have proposed an approach that considers an attacker's characteristics, such as expertise level, to adapt its deception plans.

In this paper, we present HoneyBug, an adaptive deception system for web applications that both blends seamlessly with the runtime environment, and reveals different deception plans based on attackers' expertise level and interests. HoneyBug constructs a web honeypot from an existing production web application by introducing fake vulnerabilities in its replica. Both the resulted honeypot and the real web applications are then placed behind HoneyBug. During its operation, HoneyBug monitors all the traffic, and based on its characterization model, decides whether the traffic must be redirected to the web honeypot. In this process, HoneyBug also learns attackers' expertise level and the attack vector they are interested in, and utilizes this knowledge to adapt the honeypot by enabling or disabling the introduced fake vulnerabilities.

HoneyBug is a self-adaptive system that is designed based on Monitoring, Analyzing, Planning, and Executing (MAPE) model presented in [4] and offers the following features:

- **Transparency.** One of the main disadvantages of traditional deceptive systems is their limited field of view, which means they cannot engage an attacker who is sending traffic toward them. From the users' perspective, HoneyBug is not separable from the original web application. As a result, if an attacker targets the original web application, their attack traffic will be received by HoneyBug.
- **Indistinguishability.** Based on the structure of the web application and its appearance, an attacker cannot distinguish whether they are working with the production web application or with the HoneyBug web application.
- **Adaptivity**. HoneyBug transforms the web environment by interpreting the attacker's behavior and considering their goals. This transformation occurs on the fly and in a way that will not make the attackers suspicious about it.

2. Related work

In the late '90s, the interest in defensive cyber deception picked up by the advent of honeypot systems and the formation of the Honeynet Project. In early 2000s, several seminal works such as [5, 6, 7, 8, 9] proposed different type of honeypots like *honeyd*, *honeytokens*, *honeyfile*, *dynamic honeypots*, and *honeyfarms* to enhance the security of information systems by giving insight about attackers to defenders and by diverting attackers from production systems.

Mueter et al. [10] proposed HIHAT, a generic tool to create a standalone high-interaction web honeypot from an existing web application by adding extra code to the original web application such that it logs all receiving web inputs at runtime. HoneyBug differs from HIHAT in a number of important ways. First, to create a web honeypot, HoneyBug modifies the code of the original web application to introduce fake vulnerabilities. Second, during runtime, HoneyBug characterizes attackers by observing their traffic. Based on these characterizations, it decides which of the implanted fake vulnerabilities must be activated to make the system more engaging and also to prevent honeypot mapping attacks. Last but not least, in HoneyBug, the constructed web honeypot and the original web application are hosted on web servers residing behind HoneyBug deception controller; thus, all web request destined to the original web application also passes through HoneyBug system.

Rist et al. presented Glastopf [11], which is an open-source low-interaction web honeypot that emulates different types of web vulnerabilities such as Remote File Inclusion (RFI) and Local File Inclusion (LFI) to collect data about web attackers targeting a Glastopf honeypot. It monitors the web traffic, and when detects attack requests, it emulates vulnerabilities and generates responses that the attacker expects in the case of successful exploitation. Glastopf honeypot is a standalone web application and hence suffer from the limited field of view problem; it is useless if attack traffic does not reach to the web honeypot. In addition, attackers can launch honeypot mapping attacks [12] as it is a low-interaction honeypot and has a more deterministic behavior than high-interaction honeypots. In contrast, HoneyBug is a high-interaction honeypot that creates a personalized environment for each attacker; which makes honeypot mapping attacks much more difficult if not impossible.

Recently several works such as HIDE [13] and HoneyPatch [14] have been proposed that address the limited field of view problem. HIDE uses Random Host Mutation (RHM) and K-anonymity to enhance the likelihood of trapping an attacker by increasing the chance of an attacker's encounter with honeypot systems on a network. HoneyPatch system intermingles with a production network service by wrapping around the known vulnerabilities in the service and redirecting any traffic that aims to exploit such vulnerabilities to a honeypot system.

Although HIDE can significantly reduce the field of view problem at a network level, HIDE is blind when an attacker knows the domain address of their target and uses it to reach the target, which is a typical scenario for web attackers who know the domain address of their target. HoneyPatch does not have the above limitation as it intermingles with a production network service by inserting codes around known vulnerabilities in the service and redirecting requests that attempt to exploit these guarded vulnerabilities to a honeypot system; hence HoneyPatch can reveal targeted attacks. Although HoneyPatch is proposed for network services, its idea can be extended to web applications by guarding known web vulnerabilities in them. However, according to W3Tech [15] more than 47 percent of websites are customized web applications. In order to use HoneyPatch in such web application, one must first find bugs in the application, which is a hard task, and then guard the discovered one with HoneyPatch. HoneyBug takes another approach: instead of guarding known vulnerabilities, it implants fake vulnerabilities to address the limited field of view problem.

3. Threat Model

W3Tech [15] reports that in August 2018, more than 47 percent of web applications on the Internet were not using readily available content management systems (CMS). This means many web applications on the Internet are propriety software where web attackers do not have access to the actual code. In this paper, we also assume that the attacker does not have prior knowledge about the implementation of the target web application. In other words, the application is proprietary, and the attacker does not have access to its code before launching their attack. In this way, the only way to test and find vulnerabilities on the web application is to interact with it while hosted by the defender (online scanning). In other words, the attacker cannot test the application on their side to find a vulnerability (offline scanning) and then use their knowledge, the discovered vulnerability, to launch the attack. It is worth noting that this is also a typical attack scenario on the Web.

4. HoneyBug Architecture

HoneyBug is a self-adaptive system in the sense that during runtime and based on the attacker's interactions, it adapts itself to maximize the attacker's engagement with the system; leading them to a false reality. To achieve this goal, its architecture is designed based on Monitoring, Analyzing, Planning and Executing (MAPE) model, which presented by Salehie et al. [4] for self-adaptive systems. Figure 1 illustrates the overall architecture of HoneyBug, which is comprised of the following three subsystems: (I) Deception Controller

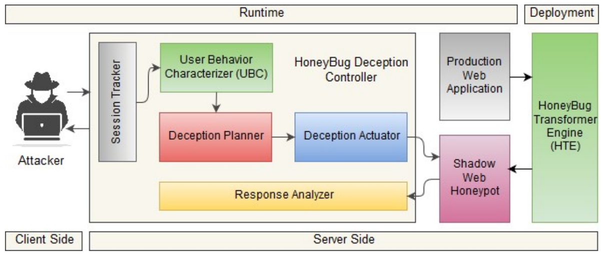


Figure 1. HoneyBug Architecture

(II) Shadow Web Honeypot, and (III) HoneyBug Transformer Engine (HTE). The first two subsystems work side by side during the operation of HoneyBug system, while the third subsystem is only used one time to construct the Shadow Web Honeypot from the original web application during HoneyBug deployment phase.

Deception controller is the main subsystem of HoneyBug and is responsible for reshaping the deception environment, i.e. the Shadow Web Honeypot, based on observed attack traffic. It sits in front of the production web server on the defender side and acts as a reverse proxy; all the traffic that are destined to the production server pass through this controller. It inspects the traffic, decides which request must be redirected to the Shadow Web Honeypot instead of the production web application. It also configures the Shadow Web Honeypot such that it would be appealing to the attacker while maintaining the inherent constraints of the production system.

Shadow Web Honeypot is the deception environment that attackers interact with. It is a variant of the original web application and is constructed by HoneyBug Transformer Engine (HTE). Both shadow and original web applications have the same web interface. However, Shadow Web Honeypot has a number of remotely controllable vulnerabilities on its back-end source code. These vulnerabilities are inactive by default, which means no one can exploit them. However, upon activation, these vulnerabilities can be exploited by attackers. Deception Controller activates or deactivates these vulnerabilities during HoneyBug operation.

4.1. HoneyBug Transformer Engine (HTE)

HoneyBug Transformer Engine (HTE) is responsible for creating a Shadow Web Honeypot from a production web application by implanting controllable vulnerabilities. The current implementation of the HTE can only transform web applications that are written in PHP, which according to W3Techs is used by more than 79.1 percent of web applications as of September 2018 [16]. A PHP application is composed of a set of PHP files, some of which receive inputs directly from the web clients. Each of these files and their associated include files can be considered as a logical subsystem of the back-end web application. HTE analyzes and transforms each of these subsystems independently from others. HTE has two components: 1. Data Flow Analysis (DFA) engine 2. Rewriter engine. In a nutshell, DFA engine statically analyzes the source code of the logical subsystems in the web application to determine the locations that vulnerabilities can be introduced. Rewriter engine modifies the code to insert controllable vulnerabilities at the identified locations. At runtime, these controllable vulnerabilities can be activated/deactivated by Deception Controller subsystem; hence modifying the vulnerability exposure of each logical subsystem on the fly.

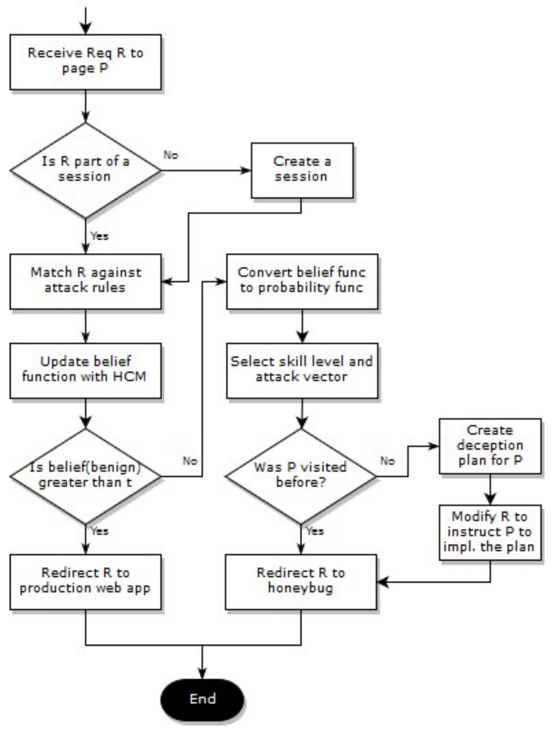
To detect the location for introducing vulnerabilities, HTE conducts data flow analysis similar to [17]. First, HTE constructs the control flow graph (CFG) of each web page in the web application. Then HTE selects those control flow paths that are started from an input variable and end up accessing one of the sink functions offered by PHP.

In PHP, all external inputs are accessed via superglobal variables such as $_GET, $_POST, and $_COOKIE. Sink functions are those built-in functions that are used by developers to interact with other external entities such as a database or local filesystem. HTE marks the conditions in the selected execution paths that are affected by built-in security-related functions in PHP. We call these conditions ”critical checkpoints”. In addition, it determines which truth value of each critical checkpoints leads to sink functions.

Security-related functions in PHP can be divided into four categories:

- **Escaping database parameters**: addslashes, pg_escape_string, pg_escape_literal, mysqli_real_escape_string, mysql_real_escape_string, mysql_escape_string, mysqli_escape_string
- **Escaping html tags**: htmlspecialchars, mb_encode_numericentity, htmlentities
- **Checking data type**: isset, empty, is_int, is_long, is_bool, is_null
- **Checking file format**: finfo_file, exif_imagetype, mime_content_type, getimagesize

HTE rewrites each of the identified critical checkpoints such that they can be short-circuited remotely. To do so, HTE generates a unique identifier for each of these checkpoints. It, then, passes this identifier along with the result of the original boolean expression inside the checkpoint, and the truth value that would bypass this security check to a function called `HoneyBug_controller`. `HoneyBug_controller` reads the value of



Receive Req R to page P
Is R part of a session
No
Create a session
Yes
Match R against attack rules
Convert belief func to probability func
Update belief function with HCM
Select skill level and attack vector
Is belief(benign) greater than t
No
Was P visited before?
No
Create deception plan for P
Yes
Yes
Modify R to instruct P to impl. the plan
Redirect R to production web app
Redirect R to honeybug
End

Figure 2. HoneyBug Deception Controller

`HoneyBugDeceptionPlanner` header field, which is inserted by Deception Controller, and decides whether to return the result of the provided boolean expression or the truth value; hence control whether the security check must be performed or bypassed.

To illustrate, suppose the generated identifier for `if(exp1)` is 123 and when *exp* is true the program halts itself with a die statement; otherwise, the control flow can reach to a sink function. In this example, HTE replaces the condition with if(HoneyBug_controller (123, exp1, false)). If the value of `HoneyBugDeceptionPlanner` header field in the HTTP request contains 123, then `HoneyBug_controller` returns false, otherwise it returns the result of the expression.

4.2. Deception Controller (DC)

Deception Controller (DC) is the central part of HoneyBug and is responsible for monitoring users' traffic, and deciding where each of the web requests must be redirected to. It forwards the requests of benign users to the original application while redirects the requests of attackers to personalized web environments that are simulated by Shadow Web Honeypot. To be more specific, when an attacker sends a request to a particular web page, DC determines the set of vulnerabilities that more likely engage that specific attacker and instructs Shadow Web Honeypot to allow only such vulnerabilities to be exploited by the attacker.

In other words, different attackers will discover different sets of vulnerabilities on Shadow Web Honeypot, based on the techniques and tactics that they have already used while interacting with the system. Figure 2 is a high-level flowchart delineating the overall process carried out by DC when a new HTTP request is received.

Deception controller has five main modules: Session Tracker, User Behavior Characterizer (UBC), Deception Planner, Deception Actuator, and Response Analyzer.

All incoming web requests to DC, first pass through Session Tracker module which marks all the requests that are originated from a specific user with a unique session id. Then, the requests are handed over to User Behavior Characterizer (UBC) module. UBC module examines each request, and reasons about the capability of the user that generated the request. To be more specific, it inspects all users' web requests with a set of generic attack detection rules, and based on the triggered rules, it characterizes the users. In current HoneyBug implementation, UBC rule set contains OWASP ModSecurity Core Rule Set (CRS) rules available in [18]. However, any security rule that is written with ModSecurity RuleSec grammar can be added to the UBC rule set.

To characterize a user, UBC utilizes a novel evidential reasoning model, which is described in details in section 5. In a nutshell, each OWASP ModSecurity rule is associated with a set of belief functions that each represents our belief about an attacker from a specific angle such as their skill level, or their interested attack vector. For each attacker, UBC characterization model combines all the sets of belief functions that are associated with the attack detection rules matched with the attacker's traffic, and comes up with a new set of belief functions. This resulted set represents our belief about the attacker based on their activity history. If UBC believes that a user is an attacker with a high probability, then it sends the request and its knowledge vector to the Deception Planner. Deception Planner has prior knowledge about the structure of the target PHP file and based on the UBC knowledge vector decides how the control flow of the PHP file should be modified. The output of the planner is a set of critical checkpoints that the actuator must relax. Actuator changes the request by inserting a new header field to instruct Shadow Web Honeypot to bypass the selected critical checkpoints.

It is worth noting that the probability distributions that are assigned to each detection rule are subjective probabilities that are assigned by an expert. It is quite plausible that a benign user is miss-identified as an attacker and migrated to the Shadow Web Honeypot if they trigger one of these detection rules by accident. As an example, suppose a user enters a comma in the last name field in the registration form as their last name is O'Brian. This comma in the input will trigger one of the detection rules for SQL injection attacks; upon updating its belief states, DC may identify this user as an attacker as the probability of being an attacker will increase drastically by the triggered rule. However, it is highly unlikely that the benign user's requests continually triggers other detection rules; as a result, after observing some requests, the system updates its belief about them and corrects its conclusion, hence redirecting them to the original application.

4.3. Shadow Web Honeypot

The client-side application of Shadow Web Honeypot is completely the same as the original application. The only difference between these two applications is the existence of controllable critical checkpoints in the back-end of Shadow Web Honeypot.

All the critical checkpoints on the original application are transformed by the HTE in a way that they can be short-circuited by Deception Planner at runtime. Each of these critical checkpoints has a control identifier and can be bypassed by HoneyBug_controller header of the HTTP request. This header contains the list of control identifiers that must be short-circuited while processing the current request. The exact mechanism to short-circuit a checkpoint is described in section 4.1.

Shadow Web Honeypot is hosted on its own web server; separated from the original web application. It interacts with its own database which is a variation of the original database. In this variation, the sensitive data is redacted, and some honeytokens [19] are added to the database. `Response analyzer` in HoneyBug Deception Controller monitors the returned responses from Shadow Web Honeypot to detect any leakage of implanted honeytokens. Upon detection of such violation, `Response analyzer` determines whether the leakage of information is caused by any of the active implanted vulnerabilities in the Shadow Web Honeypot or it is because the attacker has found a zero-day vulnerability.

5. Characterization Model

In this section, we present the HoneyBug Characterization Model (HCM) that is used by the User Behavior Characterizer (UBC) module in HoneyBug Deception Controller. This model helps UBC to gain knowledge about the attackers by observing their web requests. To be more specific, by employing HCM, UBC can answer to a set of predefined questions, such as "What is the skill level of the attacker?" and "What is the attack vector?", about each attacker.

HCM is an evidential reasoning model based on Dempster-Shafer theory (DST) [20]. In HCM model, we have a set of predefined questions Q that we are interested in knowing their answers for each attacker.

$$Q = \{q_1, q_2, ..., q_k\} \tag{1}$$

The set of all possible answers for q_i is denoted by Θ_i and is called the frame of discernment for q_i. We assume that the frame of discernment for each question represents a closed world situation which means that the set of answers is complete; the true answer to the question is a member of the frame. It should be noted that it is trivial to convert a frame of discernment with open-world assumption to a closed world one by introducing a new answer such as `other`, `unknown` to the frame of discernment to represent unknown answers.

In general, each attacker request is a source of evidence. It can give the system some clue about the correct answers to the predefined questions for that specific attacker. For example, the system can conclude that attacker x is more likely to be an expert because of the complexity of their request. However, to capture these clues and to reason about them, the model must have prior knowledge about all possible requests, which is not possible in reality. To address this problem, HCM has prior knowledge about a set of attack pattern rules, which is represented by P. It attempts to determine which rules are matched with the attacker request by using ψ function. This function returns true if the input request matches with the input attack pattern.

$$P = \{p_1, p_2, ..., p_n\} \tag{2}$$

$$\psi : R \times P \rightarrow BOOLEAN \tag{3}$$

$$req, pattern \mapsto \{true, false\} \tag{4}$$

Each of these n attack patterns is associated with a set of k mass functions. The first mass function describes a probability distribution over the power set of frame of discernment for the first question. The second mass function describes the distribution over the power set of the frame for the second question and so on. ζ is a function which is used to map a pattern to a vector that contains the corresponding k mass functions.

$$M = \{M_1, M_2, ..., M_k\} \tag{5}$$

$$M_1 = \{m_{11}, m_{12}, ..., m_{1n}\} \tag{6}$$

$$m_{ix} : 2^{\Theta_i} \rightarrow [0...1] \tag{7}$$

where Θ_i is the frame of discernment for q_i.

$$\zeta : P \rightarrow M_1 \times M_2 \times ... \times M_k \tag{8}$$

$$pattern \mapsto < m_{1x}, m_{2y}, ..., m_{kz} > \tag{9}$$

Γ is a function that returns the attack patterns that are matched with the latest r requests issued by attacker a at time t.

$$\Gamma : Number \times String \times Number \rightarrow 2^P \tag{10}$$

$$time, sessionId, count \mapsto matchedPatterns \tag{11}$$

HCM employs Yong's rule of combination [21] rather than conventional Dempster's rule of combination to combine n mass functions with the same frame of discernment but represent mutually independent bodies of evidence. The reason that we choose Yong's rule of combination is that, unlike Dempster's rule of combination, this rule considers the distances between mass functions and gives more weights to mass functions that are supported by more evidence, in this way lessen the impact of outliers.

To combine n mass functions with Yong's rule of combination, HCM, first, calculates the similarity score of any pair of mass functions m_i and m_j which is defined by

$$sim(m_i, m_j) = 1 - dist(m_i, m_j) \tag{12}$$

where $dist$ is the distance function proposed by Jousselme et al. [22] and is defined by

$$dist(m_i, m_j) = \sqrt{\frac{1}{2}(\|\vec{m}_i\|^2 + \|\vec{m}_j\|^2 - 2\langle\vec{m}_i, \vec{m}_j\rangle)} \tag{13}$$

Where $\|\vec{m}\|^2$ is $\langle\vec{m}, \vec{m}\rangle$ and $\langle\vec{m}_1, \vec{m}_2\rangle$ is the scalar product defined by

$$\langle\vec{m}_1, \vec{m}_2\rangle = \sum_{i=1}^{2^N}\sum_{i=1}^{2^N} m_1(A_i)m_2(A_j)\frac{|A_i \bigcap A_j|}{|A_i \bigcup A_j|} \tag{14}$$

Where $A_i, A_j \in P(\Theta)$ for $i, j = 1, ..., 2^N$.

Then, HCM calculates the support degree for each of the mass function which can be calculated by

$$sup(m_i) = \sum_{i=1, j\neq i}^{n} sim(m_i, m_j) \tag{15}$$

Then the credibility of each mass function is calculated by

$$crd_i = \frac{sup(m_i)}{\sum_{i=1}^{n} sup(m_i)} \tag{16}$$

Finally, HCM calculates the combination of mass functions by Yong's rule of combination as defined below:

$$m_{BOA} = \sum_{i=1}^{n}(crd_i \times m_i) \tag{17}$$

In HoneyBug, to decide about the set of active vulnerabilities on a web page, Deception Controller needs to determine the answer to the following questions about the attacker: (I) Skillfulness: what is the skill level of the attacker, and (II) Attack vector: what is the attack vector that the attacker is using. For each generic attack detection rule, we have subjectively defined the mass function for each of these questions based on expert knowledge. For the first question, the frame of discernment contains the following four answers: (I) elite, (II) intermediate, (III) script kiddie, (IV) benign.

6. Deception planner

HCM represents the beliefs that the system is reached about the attacker while observing their attack traffic. In order to decide which vulnerabilities on a requested page should be enabled, deception planner converts the calculated belief functions to probability functions by using the pignistic transformation presented in [23].

$$P_m(A) = \sum_{\phi \neq B \subseteq 2^{\Theta}} m(B) \frac{|A \cap B|}{|B|}, \forall A \subseteq \Theta \qquad (18)$$

Then, deception planner randomly selects the answer for each of the question based on the corresponding probability function $P_m(A)$. At this point, DC concludes that the attack vector of interest is x, and the attacker type is y.

For each page, deception planner knows which control paths can lead to what kind of vulnerabilities, and for each of such paths, it knows the controllable checkpoints and their strength score. This information is created by HTE module while analyzing the original application. Deception planner determines the PHP file that is the target of the attacker's request. Then, it selects all control paths that can be used by the attacker to launch their attack by using a specific attack vector such as XSS that the system believes that the attacker is currently testing. For each of such paths, the controller selects the most difficult checkpoint that has not been already bypassed. If the total difficulty level of the remaining checkpoints is below a given threshold, the planner discards the selection. Otherwise, it will add its identifier to HoneyBug_controller header of the request.

7. Evaluation

In this section, we present our evaluation of HoneyBug system in an experimental setting. In HoneyBug, our primary goal is to maximize the engagement of attackers with HoneyBug system. To achieve this goal, the system must be attractive in the sense that it must exhibit vulnerabilities that can be exploited by the attacker. It also must adapt itself when the attacker changes their attack vector. As we do not know attackers beforehand, the system needs to make decisions in an online fashion and in a fraction of seconds to prevent differential timing attacks. In the rest of this section, we investigate these properties by conducting several experiments. Each experiment was repeated $10,000$ times, and their averages were reported.

7.1. Attractiveness

To increase the attractiveness of Shadow Web Honeypot, HoneyBug must enable those vulnerabilities in the web honeypot that can be exploited by the attacker. These vulnerabilities should appear as early as possible before the attacker gives up examining the web application. To measure attractiveness, we define attractiveness score A which is defined in Eq. 19:

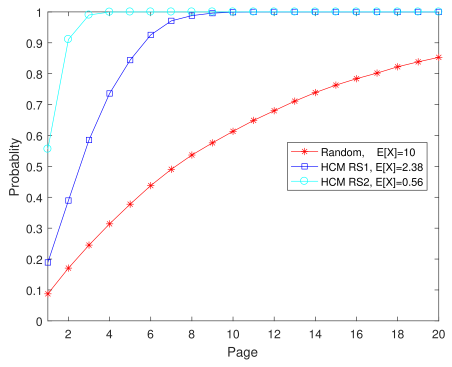


Figure 3. The probability of observing a vulnerability x of interest after visiting n pages when (I) selecting vulnerabilities randomly (II) selecting vulnerabilities based on the knowledge that is gained from HCM.

$$A = max(0, 1 - \frac{E[X]}{|pages|}) \qquad (19)$$

Where X is the number of pages that are visited by an attacker before reaching a page with a vulnerability that they can exploit, and $E(X)$ is the expected value of X. The attractiveness score is a real number between 0 and 1 inclusive. If the attractiveness score of a system is 1, it means attackers can find a suitable vulnerability on the first page that they visit. However, in reality, where the system does not know the attacker beforehand and therefore does not have any clue about the kind of vulnerabilities that the attacker is interested in, the system cannot reach this maximum score unless it enables all vulnerabilities on all pages which makes it too vulnerable and hence too suspicious.

In HoneyBug, deception planner uses HCM output to determine the type of vulnerabilities that the attacker is looking for, and enables such vulnerabilities in the Shadow Web Honeypot. As we mentioned in section 5, HCM is an evidential reasoning model that enables HoneyBug to gain knowledge about the attacker by observing their interactions with the system. An alternative strategy that deception planner can use is to randomly enable vulnerabilities on the target shadow web pages.

Figure 3 shows the cumulative distribution function (CDF) of finding a particular vulnerability type t, out of 10 vulnerability types, for the first time when visiting n pages in a web application with twenty web pages when deception planner activates implanted vulnerabilities with the strategies mentioned above.

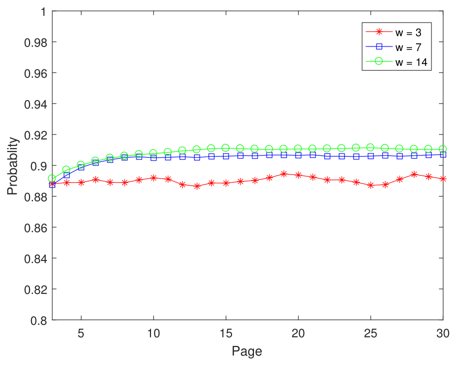


Figure 4. The effect of request windows size on the convergence of HCM model toward the true answer

Each web page in this application contains all the ten types of vulnerabilities, which can be activated by the deception planner on demand. In this experiment, only one type of vulnerabilities can be activated on each page; the planner can also decide to keep all the vulnerabilities deactivated in a web page. Therefore, the planner has 11 options for each page in the web application. In random strategy, the planner chooses type t with probability $p(t) = 1/11$. In HCM RS1 strategy, we assume attack traffic matches with attack detection rules that are associated with belief functions that assign mass value $1/11$ to "vulnerability type t is the vulnerability of interest" ($m(t) = 1/11$) and mass value $1 - 1/11$ to all answer ($m(\Theta) = 1 - 1/11$), which means that the rules' ignorance about the true answer is 0.91. In HCM RS2 strategy, the rules are more concrete ($m(t) = 0.5$ and $m(\Theta) = 0.5$); the ignorance level is 0.5.

As it is shown in figure 3, when the deception planner makes decision-based on the output of HCM model, the attacker finds, with higher probability, the vulnerability of interest earlier than when it selects vulnerabilities randomly. In other words, the expected value of X, the number of observed web pages before finding a suitable vulnerability, is lower in case of relying on HCM outputs; the resulted honeypot will have a higher attractiveness score. In addition, when the attack detection rules can give more evidence about the true answer (i.e., more precisely they can provide information about the attack vector that the attacker is looking for), the HCM output converges more rapidly.

7.2. Adaptiveness

Attackers may change their attack vectors during their operations. Therefore, deception controller must be able to capture these shifts and re-adapt itself accordingly. HoneyBug deception controller uses HCM model, a DST model, to gain knowledge about the attacker. Traditionally, Dempster's rule of combination [24] is used to combine bodies of evidence; we also used it in our early implementation of HCM.

However, in our experiments, we observed that most of the time when the attacker, after a few steps, changes their attack vector, the HCM model could not re-converge to the new answer. In such cases, the belief function about the attack vector that HCM calculated before the change is in complete conflict with the mass function raised after the change; which makes their combination, with Dempster's rule of combination, undefined. Through experimentation, we found Yong's rule of combination more flexible in combining a set of mass functions with conflict as it assigns a weight to each of them based on their similarities and averages them. As a result, outliers will have lower weights and thus have a lesser influence. Moreover, HCM only considers attack detection rules that are raised by the last w requests and combine their associated mass functions. As a result, aged requests will not be considered in characterizing the attacker.

In our current implementation, HCM has 58 attack detection rules that are taken from ModSecurity CRS and its request window size is 14. For each of these rules, we defined two mass functions one to determine skillfulness and one for the attack vector. HCM rule set contains 21 SQL injection, 22 Cross-Site Scripting (XSS), 4 Remote File Injection, 2 Local File Injection, and 9 Unix Command Expression attack detection rules. Figure 4 shows the effect of request windows size on the convergence of HCM model toward the true answer. In these experiments, only attack detection rules for a specific attack vector are raised. As the window size increases the HCM model converges towards the result sooner, and it is more steady. Figure 5 shows how HCM re-adapt its belief when an attacker changes their attack vector on page 14 from SQL injection to XSS. As it is shown in the figure, after a few pages, the HCM model captures the changes that occurred on page 14. During their operation, an attacker may also raise some detection rules that are associated with attack vectors that are not in their interest. Figures 6 shows how HCM readjust if 10 percent of the raised attack rules do not represent the attack vector that the attacker is interested in.

7.3. Scalability

In this section, we evaluate the time and space overheads that are imposed by HoneyBug and show that its overheads are minimal and hence it can be employed in a real environment to protect a real web application.

HoneyBug adds a custom header field to HTTP requests forwarded to Shadow Web Honeypot to instruct the honeypot to bypass a subset of its critical checkpoints. The field value of the inserted field is a

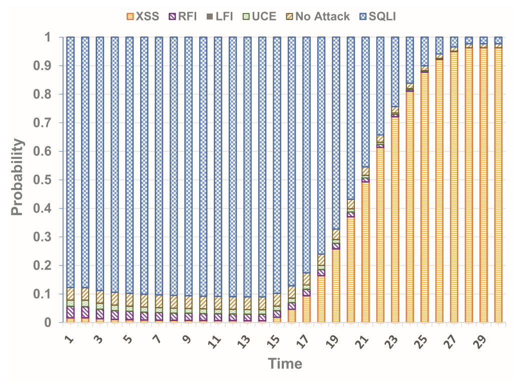


Figure 5. Convergence of HCM when the attack vector is changed from SQLI to XSS at time 14.

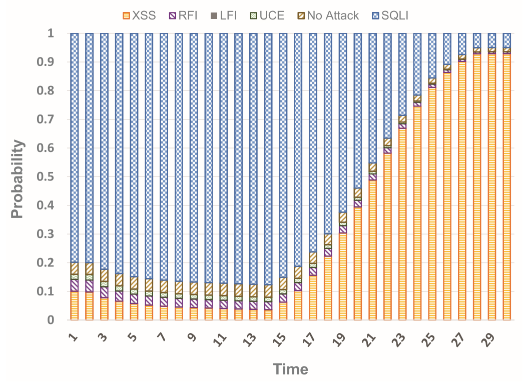


Figure 6. Convergence of HCM when the attack vector is changed from SQLI to XSS at time 14. Attacker chooses other attack vectors with 0.1 probability

list of critical checkpoint identifiers that are separated by commas. The length of each identifier is fixed and is 16 characters. The size of this field in a request depends on the number of critical checkpoints that the deception control demands to be bypassed by web honeypot while processing the request. Although there is no limitation on the size of HTTP header field, a modern web server, such as Apache HTTP Server and IIS, sets an upper limit on the length of a header field. For instance, in Apache HTTP Server v2.4, the limit is 8KB [25].

As we discussed in Section 4, HoneyBug has two main subsystems which work with each other at runtime: Shadow Web Honeypot and Deception Controller (DC). Shadow Web Honeypot is constructed from a production web application. The main difference between these two web applications is that in Shadow Web Honeypot, some of the security checkings can be bypassed. As a result, the performance of Shadow Web Honeypot is quite similar to the production web application. DC acts as a reverse proxy and processes all the requests sent by users. The main processing in DC takes place in UBC module. Figure 7 shows the time that

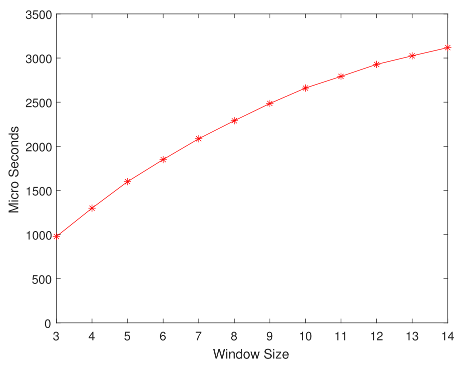


Figure 7. The impact of windows size on rule combination processing time

HCM model requires to combine n mass function using Yong's rule of combination. It is worth noting that the delay introduced by DC affects all benign and malicious requests; thus, it cannot be directly used to determine whether a request is being redirected to Shadow Web Honeypot.

8. Scope and Limitations

In this section, we discuss a few limitations of HoneyBug. First, the accuracy of HoneyBug is bound to the accuracy of its embedded detection rules. For example, if an attacker sends malicious requests that do not trigger any of the detection rules, then HoneyBug concludes that the user is benign. Second, an informed attacker can attempt to remain on the production application by interleaving malicious requests among a great number of benign requests to reduce the probability that HoneyBug UBC marks them as an attacker; nevertheless, this will prolong the scanning process. Third, an attacker may discover a zero-day vulnerability when examining the Shadow Web Honeypot; however, they cannot distinguish such a vulnerability from the fake ones. The attacker can act as a benign user for a while to be redirected to the production web application and then test the discovered vulnerability to determine whether it is real. Since with a high probability, the discovered vulnerability is fake, this strategy significantly slows down the attacker. Moreover, if the attacker exploits the zero-day vulnerability while they are on the Shadow Web Honeypot, then the deception controller may detect the exploit as the data stored on the Shadow database contains honeytokens. Finally, in the current implementation, public data in the production database is replicated with the Shadow database. During session redirection, stored information about the user, annotated by developers, is also migrated to the Shadow database.

However, no temporary data, stored in the memory, is transfered between the production and shadow systems. Such capability can be implemented by utilizing the approach mentioned in [14]. In the future, we plan to investigate different approaches that can be employed to reduce the discrepancy between the two systems at the data level.

9. Conclusion

In this paper, we present HoneyBug, a novel adaptive deception system for web applications. HoneyBug is based on MAPE (monitor, analyze, plan, execute) architecture [4] that changes its deception plans based on the characterization of attackers, which makes it immune against honeypot fingerprinting attacks as the deception environment dynamically changes from one attacker to another. The HoneyBug Characterization Model (HCM) enables HoneyBug to attribute the attackers by observing their interactions. In addition, the HoneyBug Planner changes the control flow of the application to select the most appropriate exploit path for attackers. Our evaluation shows that HoneyBug exhibits high attractiveness score based on our metrics. It also shows that HoneyBug can scale to handle a large number of simultaneous request and attackers as it exhibits a low-performance overhead. In the future, we plan to further investigate the effectiveness of HoneyBug system against human attackers through red teaming experiments and also deploy it in the wild to evaluate its effectiveness against real-world attackers.

References

[1] N. Provos, "Honeyd-a virtual honeypot daemon," in *10th DFN-CERT Workshop, Hamburg, Germany*, vol. 2, p. 4, 2003.

[2] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling, "The nepenthes platform: An efficient approach to collect malware," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 165–184, Springer, 2006.

[3] I. Kuwatly, M. Sraj, Z. Al Masri, and H. Artail, "A dynamic honeypot design for intrusion detection," in *Pervasive Services, 2004. ICPS 2004. Proceedings. The IEEE/ACS International Conference on*, pp. 95–104, IEEE, 2004.

[4] M. Salehie and L. Tahvildari, "Towards a goal-driven approach to action selection in self-adaptive software," *Software: Practice and Experience*, vol. 42, no. 2, pp. 211–233, 2012.

[5] L. Spitzner, "Honeypots: Catching the insider threat," in *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pp. 170–179, IEEE, 2003.

[6] J. Yuill, M. Zappe, D. Denning, and F. Feer, "Honeyfiles: deceptive files for intrusion detection," in *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*, pp. 116–122, IEEE, 2004.

[7] N. Provos *et al.*, "A virtual honeypot framework.," in *USENIX Security Symposium*, vol. 173, pp. 1–14, 2004.

[8] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling, "The nepenthes platform: An efficient approach to collect malware," in *Recent Advances in Intrusion Detection*, pp. 165–184, Springer, 2006.

[9] L. Spitzner, "Honeytokens: The other honeypot. 2003," *Internet: http://www.securityfocus.com/infocus/1713*, 2006.

[10] M. Mueter, F. Freiling, T. Holz, and J. Matthews, "A generic toolkit for converting web applications into high-interaction honeypots," *University of Mannheim*, vol. 280, pp. 6–1, 2008.

[11] L. Rist, S. Vetsch, M. Kossin, and M. Mauer, "Know your tools: Glastopf-a dynamic, low-interaction web application honeypot," *The Honeynet Project*, vol. 4, 2010.

[12] V. Yegneswaran, C. Alfeld, P. Barford, and J.-Y. Cai, "Camouflaging honeynets," in *IEEE Global Internet Symposium, 2007*, pp. 49–54, IEEE, 2007.

[13] J. H. Jafarian, A. Niakanlahiji, E. Al-Shaer, and Q. Duan, "Multi-dimensional host identity anonymization for defeating skilled attackers," in *Proceedings of the 2016 ACM Workshop on Moving Target Defense*, pp. 47–58, ACM, 2016.

[14] F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser, "From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 942–953, ACM, 2014.

[15] W3Techs, "Usage statistics and market share of content management systems for websites," 2018.

[16] W3Techs, "Usage statistics and market share of php for websites," 2018.

[17] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Security and Privacy, 2006 IEEE Symposium on*, pp. 6–pp, IEEE, 2006.

[18] OWASP, "Owasp modesecurity core rule set (crs)," 2017.

[19] L. Spitzner, "Honeytokens: The other honeypot," 2003.

[20] G. Shafer, "Dempster-shafer theory," *Encyclopedia of artificial intelligence*, pp. 330–331, 1992.

[21] D. Yong, S. WenKang, Z. ZhenFu, and L. Qi, "Combining belief functions based on distance of evidence," *Decision support systems*, vol. 38, no. 3, pp. 489–493, 2004.

[22] A.-L. Jousselme, D. Grenier, and É. Bossé, "A new distance between two bodies of evidence," *Information fusion*, vol. 2, no. 2, pp. 91–101, 2001.

[23] P. Smets, "Data fusion in the transferable belief model," in *Information Fusion, 2000. FUSIon 2000. Proceedings of the Third International Conference on*, vol. 1, pp. PS21–PS33, IEEE, 2000.

[24] K. Sentz and S. Ferson, "Combination of evidence in dempster-shafer theory," tech. rep., Sandia National Labs., Albuquerque, NM (US); Sandia National Labs., Livermore, CA (US), 2002.

[25] T. A. S. Foundation, "Apache http server version 2.4 documentation," 2017.