A Review on Artificial Intelligence Techniques for Developing Intelligent Honeypot

Wira Zanoramy Ansiry Zakaria
Computer Systems and Technology
Faculty of Computer Science and Information Technology
University of Malaya
50603 Lembah Pantai
Kuala Lumpur, Malaysia
wirazanoramy@gmail.com

Miss Laiha Mat Kiah
Computer Systems and Technology
Faculty of Computer Science and Information Technology
University of Malaya
50603 Lembah Pantai
Kuala Lumpur, Malaysia
misslaiha@um.edu.my

Abstract—**Honeypot is a closely monitored computer resource that emulates behaviors of production host within a network in order to lure and attract the attackers. The workability and effectiveness of a deployed honeypot depends on its technical configuration. Since honeypot is a resource that is intentionally made attractive to the attackers, it is crucial to make it intelligent and self-manageable. This research reviews at artificial intelligence techniques such as expert system and case-based reasoning, in order to build an intelligent honeypot.**

Keywords - honeypot; intelligent system; knowledge-based system; case-based reasoning; expert system

I. INTRODUCTION

The term 'honeypot' is usually being used for representing 'a container filled of honey', which is often playing off the image of nice sweetness that is being used as an attractive trap for bears. But in the case of network security, this term is being used to represent a security technology that is based on deception. A honeypot is defined as a security resource whose value lies in being probed, attacked or compromised [1]. When we look back into history, the idea behind honeypot is based on the works of Sun Tzu and Clifford Stoll [2].

Honeypots are resources that are purposely being setup in the network in order to lure the attackers and at the same time captures their tools and techniques without the attacker's consent. The information captured are then will be used to learn about the attacks and as a guide to improve security measures in the future [2].

Honeypots are resources that are meant to have no authorized activity and production value on it. By default, a honeypot should not be receiving any interactions. The reason behind this is to make a clear assumption that: any interactions captured by the honeypot are considered malicious. Any detected activities on the honeypots are assumed to be a probe, scan, or attack. The value of a honeypot comes from their ability to capture such activities.

Honeypots are camouflaged inside the network like any other production hosts, but in reality, they are fake hosts that are pretending like production hosts. This is to attract the attackers into exploiting the honeypots instead of the legitimate production hosts.

Honeypot can be built in many forms, either in the form of physical machine, virtual machine (VM) or emulated virtual host. A physical honeypot is a real computing platform that has its own valid IP address. For example, a computer installed with Fedora Linux or Windows 7 with a running network services, like FTP, Telnet or SMTP.

Honeypot VM can be built by using virtualization software like VMWare Workstation, Qemu-KVM, Virtualbox, Parallels Desktop or User Mode Linux (UML). By utilizing either of these tools, honeypot VM can be created with any type of operating system. This software is installed on a computing platform and user can create one or multiple VM(s) running on the same platform.

Honeypot also can be built in the form of emulated virtual hosts. By using tool like Honeyd, we can emulate up to thousands of virtual hosts running different types of operating system on top of a single machine. Each of these virtual hosts is configured with a certain behavior and personality, in which it defines how the virtual host will respond to attackers' interactions. For example a virtual host can be programmed to contain a Perl script that is emulating a Sendmail service.

II. TYPES OF HONEYPOT

Honeypots are classified based on its level of interactions. The word 'interaction' here means - the degree of communications that are being allowed for the attacker to exploit the honeypot. At the same time, it also means the other way around; the 'level of interaction' defines how deep a honeypot is allowed to be exploited by an attacker. The more an attacker can do to a honeypot, the more information can be collected by the honeypot from the attacker, however the more risk the honeypot will most likely has [3]. There are two types of honeypot: low-interaction and high-interaction honeypot [4].

This research is funded by UM Research Grant (UMRG).

A. *Low-interaction Honeypot*

Low-interaction honeypot has the lowest interaction capability with an attacker and it is also the simplest honeypot to setup. This type of honeypot only provides minimal services and usually it is in the form of virtual host with emulated services. For example, a virtual host running an emulated FTP service, built by using honeypot framework software called Honeyd.

Honeyd is a popular open source low-interaction honeypot framework that offers a simple way to emulate virtual hosts on a single machine [5]. The main advantage of low-interaction honeypot is it has low risk: because it only offers emulated services to the attacker. The attacker's action is limited to what is being offered or emulated within the virtual host, and they can only scan and connect to the offered ports [6]. Another advantage of low-interaction honeypot is, it is easy to deploy and maintain. Its disadvantage is the amount of information that can be collected by the virtual host is small.

Honeyd is built in the form of UNIX daemon and can be run on all UNIX-based and BSD platforms. This tool is open source software released under the GNU General Public License. Honeyd is a small daemon that has the ability to create and deploy low-interaction virtual honeypots around the network. All these virtual honeypots can be configured to run certain services and system administrator has the freedom to set the personality of each virtual honeypot. The appearance of each single virtual honeypot can be customized so it can be set to mirror the current network environment. In the eyes of attackers, they will see the virtual honeypot as a usual production host appears to be running certain operating systems with some services [6].

Honeyd has the ability to emulate different types of operating systems at the IP layer level and run scripts attached to specific ports using *stdin* and *stdout*. Instead of being limited to one IP address, Honeyd has the ability to bind to multiple IP address, making it able to create many virtual honeypots that emulate multiple hosts with different operating systems and services, and map them to unused IP addresses. Honeyd is very practical in mimicking operating system TCP/IP stacks and offering services for the attackers to probe and interact [7]. Honeyd works on the concept that when it receives any interactions for a host that does not exist, it assumes that the interaction attempt is an attack and at the same time Honeyd will log the activity [6]. Honeyd will start the emulated services for the port on which it accepts the interactions. This emulated service will continuously communicating with the attacker and logs all activities. After the attacker finish his interaction, the emulated service will stop. These same steps will keep on repeating when Honeyd accepts any probe for a non-existent host. Sadasivam *et al.* has written a paper regarding setting up Honeyd for accomplishing some example honeypot projects [6]. Other examples of low-interaction honeypots are HoneyBOT [8], Dionaea [9].

B. *High-interaction Honeypot*

High-interaction honeypot is a complex honeypot solution because it offers a setup of real operating system or a suite of real services to attackers in which nothing is emulated or restricted [10]. The high-interaction capability enables security practitioners to collect extensive amount of information from the attacker's malicious activities. This information can be used to study clearly the attacker's behavior, tools, motives and even their identity. The ability to gather huge amount of data is the main advantage of high-interaction honeypots. Setting up this type of honeypot is time-consuming and it is also hard to maintain [6], [11]. Security practitioners need to have a certain level of expertise and experience in order to run this type of honeypot. Honeynet is an example of high-interaction honeypot. Honeynet is a network of two or more honeypots that works together in deceiving and trapping the attackers.

III. THE ISSUES OF HONEYPOT DEPLOYMENT

The effectiveness of security technologies such as intrusion detection system (IDS), intrusion prevention system (IPS), anti-virus, encryption keys, anti-phishing tool, and firewall heavily relies on human configuration and maintenance effort. Like any other security technologies, honeypot also face the same problem. It also needs proper configurations and timely maintenance. In order to deploy honeypots, there are two areas of challenges that have been identified – to configure it and to maintain it.

In setting up a honeypot, we need to configure it correctly. Since honeypot works based on the concept of deception, its appearance to the attackers is the most crucial thing to be taken care of before it can be deployed in the network. Error in configuring it, can lead to missed detection or even not attracting the attackers at all. This will void the original purpose of the honeypot. Configuration issues like what type of OS personality that the honeypot will have, how many TCP and UDP ports to open, which services to offer, at which IP address the honeypot will be located, how the honeypot will response to the attacker and so on.

Nowadays, there are many types of computing devices, operating systems and services that populates the network environment of any organization. Operating system vendors keep on introducing new versions of its OS. The users of mobile computers usually plugs in and out of the network from time to time. High performance computing platform like the servers and workstations are sometimes being introduced or even upgraded. Outdated computing technologies are most likely being phased out from the network environment. This is a scenario to show that the honeypot setup needs to be as dynamic as its environment in order to cater the changing nature inside the network. Honeypots must have a dynamic and adaptive feature, where it can handle and maintain itself in situations like this throughout the times.

IV. RELATED WORKS

Lance Spitzner, from The Honeynet Project, has addressed some of the disadvantages of deploying static honeypots [12]. In his paper, he proposed a new concept for honeypots, named as ‘dynamic honeypot’ in order to cater all the problems in configuring and maintaining the honeypots. According to

Spitzner, dynamic honeypots is a plug and play solution, which can automatically learn about the network environment and manage all the honeypots configurations autonomously. The ability of dynamic honeypots could address the trouble of a lack of resources for honeypots configurations and the problem of detection. The concept of dynamic honeypots is proposed to reduce the amount of configuration and maintenance needed and potentially decrease the possibility that attackers would easily detect a honeypot.

Dynamic honeypot is envisioned to have the ability to learn about the network environment and then deploy honeypots to appropriately blend in with the current snapshot of the network. After the deployment, the dynamic honeypot will continuously monitor the network for any changes and it will update the current honeypot configurations based on the changes. For example, if a network has all Windows-based hosts, the dynamic honeypots will autonomously deploy Windows honeypots. If suddenly, UNIX-based host is being added to the network, automatically UNIX honeypots will be deployed. As proposed by Spitzner, dynamic honeypot is like an appliance that can be simply plugged into the network, it learns the environment and then it will deploy proper number of honeypots with accurate configurations. Besides that, it will also learn and adapt to any changes inside the network.

Basically, a dynamic honeypot consists of three main modules: remote host fingerprinter, honeypot configuration builder and honeypot handler. The remote host fingerprinter is used to scan for remote host's information. Honeypot configuration builder is the part that configures the honeypot settings and behavior. The honeypot handler is used to deploy honeypot based on the configurations made by the earlier module.

Spitzner proposed that for the remote host fingerprinter module, we should use passive fingerprinting technique [12]. This is to avoid introducing more traffic to the network. Applying active fingerprinting also could bring the whole network down and consumes bandwidth.

Kuwatly *et al.* used different approach in doing the fingerprinting part, where they integrated both fingerprinting techniques: active and passive [13]. Their version of dynamic honeypots uses both techniques interchangeably based on some predetermined conditions. In [13], they proposed a dynamic honeypots design for the purpose of detecting intrusions. Even though their research in dynamic honeypots is more towards catering intrusion detection, their setup on dynamic honeypots is relevant to be discussed here. The dynamic honeypots design that they proposed is a little bit complex. This is because they have added about three more supporting components to it. They integrated a number of components to become the 'dynamic honeypot server'. For the information gathering part, they have four subcomponents: Nmap, Xprobe, Snort and p0f. This part directly and dynamically controls the virtual honeypots configurations under Honeyd. The basic construct of their approach is mostly based on Spitzner's idea. The parts that made their contribution is different from Spitzner's original idea on dynamic honeypots consists three components: a database that consists of hosts' information, a dynamic honeypot engine which handles the task of managing the virtual honeypots identity configurations, and a real-time visual correlation infrastructure that aid administrators in interfacing with the dynamic honeypots server.

Jiang & Xu proposed the idea of catering honeypots architecture called BAIT-TRAP [10]. It has the capability of managing the personality of each honeypots by offering the services that the attackers currently aiming for. BAIT-TRAP functions by constantly monitoring the network traffic, identifies services that are currently attractive to the attackers and deploy honeypots that runs such services. These services are meant to be the bait to attract the attackers to interact with the honeypots. During real implementation of BAIT-TRAP inside Internet environment, they managed to capture a number of trendy attack incidents [10].

Leita *et al.* have developed an add-on tool for Honeyd called ScriptGen [5]. It is designed to be completely autonomous and allows the emulation of any protocol of any kind without any knowledge about it. This tool has the ability to automatically generate Honeyd configuration scripts. They analyzed the quality of the generated scripts by launching known attacks on the machines that runs the generated scripts. Besides that, they also deployed the same machine on the Internet by putting it next to a high-interaction honeypots for about two months. Their approach in doing this is much more complicated because it involves the skills and knowledge of building state machines.

Hecker *et al.* built a Honeyd Configuration Manager in Perl [14]. This module actively scans the network using active OS fingerprinting tool – Nmap. Even though this approach does disturb the network with bandwidth consumption, Nmap can detect the state of ports from a remote host faster than the passive OS fingerprinting tool. The module triggers Nmap to use SYN stealth scan, RPC scan, UDP scan and OS detection towards the remote hosts. For the IP address assignment, it is not being done automatically. In their approach, the administrators still need to make some additional configuration to the generated Honeyd script.

Liu *et al.* designed a dynamic honeypot system with a different approach [15]. Instead of using either active or passive OS fingerprinting technique, they combined both of it. They applied active OS fingerprinting technique with a combination of passive technique. This is to ensure that the remote OS detection result is as accurate as possible. Their dynamic honeypot system is also utilizing the Honeyd tool as the honeypot handler, same as [5] and [14].

Chowdhary *et al.* and Wagener *et al.* uses machine learning technique as an approach for implementing an intelligent mechanism in dynamic honeypot [16], [17]. Chowdhary *et al.* proposed a twist of data-mining, in which they called it as 'service-mining' [16]. By using this approach, they are able to learn about the behavior of the real services inside the network. The behavior is extracted from the interactions that are being produced by the real services. Later on, an emulated version of the service is created based on the learned behavior. These emulated services behave just like its counterparts: the real services and it is deployed on a honeypot, to distract the attackers from the real services.

V. Proposed Approach

The ultimate goal of AI is to build information systems that are capable of emulating human reasoning in making decisions to solve real world problems. Intelligent system is defined as a software that is built with AI capabilities. In this paper, we reviewed two AI techniques: expert system and case-based reasoning.

A. Expert System

Expert system (ES) is a computer program that simulates the way of a human expert in solving problems for a specific domain. It is one of the most implemented artificial intelligence techniques in the world. It is designed to solve domain specific problems by running a 'reasoning' process like a human expert. One of the earliest expert system, MYCIN, was developed in the 1970s at Stanford University. MYCIN's functionality is to help physicians to diagnose bacterial infections among patients. The system also has the capability to recommend therapy and dosage of antibiotics based on the patient's body weight. Another example of early expert system is DENDRAL. Its task was to assist organic chemists in unknown organic molecules identification.

Basically, an ES consists of a working memory (WM), knowledge-base (KB) and an inference engine (IE). The working memory is like a temporary buffer to hold the facts which is given by the user from the real world. These facts will be processed by the IE. This engine will utilize the rules inside the knowledge-base in order to find solution. In this reasoning process, the related rules will be fired from the KB until a solution is finally formulated. The rules in the KB are knowledge gathered from domain expert or related literatures that have been coded in the form of IF-THEN representation. An example rule is shown below:

```
IF the light is green THEN go;
IF the light is red THEN halt;
...
```

Building an ES is pretty straight forward. Nowadays, with the existence of ES shells, it is quite easy to rapidly develop an ES. The ES developer can focus more of his effort in building the KB. ES shells like the C Language Integrated Production System (CLIPS) [18] and Java Expert System Shell (JESS) [19] to name a few, are the most suitable tool to build it. Both of these tools have strong and active community of users and developers. Since both of it are based on popular programming languages, C and Java respectively, it is easy for an ES builder to integrate his ES with other interfacing applications, like a graphical user interface (GUI) or even a web service.

The biggest challenge in building an expert system is to build its KB. A perfect KB relies on the skills of the knowledge engineer or ES developer in gathering knowledge from the domain expert and later programmed it in the form of rules.

B. Case-based Reasoning

Case-based reasoning (CBR) is basically a way of solving problem by reusing past solutions. It is one of artificial intelligence method in solving problems. It solves a new problem by reusing the solution of a past similar problem. In ES, the knowledge is coded in the form of rules. But for CBR system, the 'knowledge' is originated from 'past experience' and it is coded in the form of cases. A case is a knowledge model for a particular experience in a particular domain. A case consists of two parts: problem description and solution. The 'problem description' part contains the information about a problem situation in the past. Meanwhile, the 'solution' part contains the information on the solution for that particular problem. In other words, a case contains information about past event and past solution. Cases are stored inside the persistent memory (for example: database), so it is easy for later retrieval. According to [20], CBR systems solve new problems by adapting solutions that were used to solve old problems. The basic idea in CBR is it tries to mimic how human being handles a problem that challenges them. We usually, use our memory to find any related past occurrences of the similar problem, and tries to adjust the previous solution so that it can fit into the current situation (problem). Basically, there are four steps involved in a CBR cycle: retrieve, reuse, revise and retain. It is usually abbreviated as – the 4R(s).

i. **Retrieve**: In this step, the features inside the current problem will be match with the most similar past cases inside the case storage. Past cases that are most relevant to the current problem will be retrieved from the storage. Usually this step involves the calculation of the degree of similarity between the current problem and the stored past problems. The past case that has the nearest similarity will be the winner. K-nearest neighbour (k-NN) algorithm is an example of method that is being used in this step.

ii. **Reuse and/or Revise**: In this step, if necessary, the value of some or all of the features of the solution part of the past problem is altered in order to fit with the current problem. In this phase, the algorithm involved is very domain-dependent. Examples of adaptation techniques are substitution and parameter adjustment. The newly formed solution in this step is then fit into the current problem to solve the situation in question.

iii. **Retain (store)**: In this step, it is said that the CBR system 'learns a new experience'. The newly formulated solution is then combined with the descriptions of the current problem in question, and then it is stored inside the memory or case-base for future usage. Since the CBR system gained 'new' experience in this step, from time to time, the system will become more efficient and less resource consuming when it encounters a similar problem or a nearly-similar ones in the future. In some cases, this step also involves case maintenance, where the past cases inside the case-base that are seldom being retrieved are deleted. This is to make sure that the storage is efficiently managed and positively affects the time in retrieving a case in the next CBR cycle. An optimized case-base also affects the system performance in solving problems.

There are many open source tools available for building a CBR system, such as JColibri [21], Indiana University Case Based Reasoning Framework (IUCBRF) [22] and myCBR [23]. Among these three tools, JColibri is the most updated, supported and has an active community of CBR system developers. One of its variant, the JColibri Studio, is easily integrated into Eclipse and it comes with additional tools for building rapid CBR system.

VI. Discussion

Both techniques, CBR and ES have their own strengths and weaknesses in building intelligent system for the domain of honeypot configuration. In the case of CBR and ES, both of it need a problem to solve. For this research, the problem is gathered from the network environment using network fingerprinting tools like Nmap or p0f. These tools will capture the needed network host's information and input it into the AI engine. For example, the information collected from the host is operating system type, IP address, active port numbers and uptime. The AI engine will do some reasoning on this information to produce solution.

The ability to learn and adapt in the CBR cycle is the main reason why this technique would be suitable for building intelligent honeypot. Because honeypot configuration is a complex and not yet a fully understood domain, it is easier to build the casebase. We just need to find some sample cases from the real world, code it in into the casebase and let the CBR adaptation utilize it.

Since CBR merely depends on learning from past experiences or past cases, there is no need to perform thorough knowledge engineering process like in building the knowledge-base for the ES. Figure 1 shows a structure for case representation.

In the case of building an ES for the domain of honeypot configuration, the KB must contain the rules that representing the knowledge and skills in configuring honeypot. This knowledge can be gathered from domain experts or even literature sources like books and manuals.

The 'IF' part of the rules, should contain the features of the detected production host in the network subnet in view. While for the 'THEN' part, should contain the information on honeypot configuration. The ES developer must code all knowledge gathered from the domain expert into the KB. By reasoning from the rules, ES will have the capability to determine what type of OS, which IP to deploy and what ports to open inside the honeypot configuration. Figure 2 shows some sample rules for honeypot configuration that is stored inside the KB. Later the developer must decide which reasoning approach that he wants the IE to work; either it is forward-chaining [24] or backward-chaining [25]. Forward-chaining is usually called as data-driven method. This is because; in forward-chaining the rules are triggered based on the supplied data or facts.

Problem Description
Detected host
IP address
Uptime
TCP, UDP, ICMP ports status
Solution
OS Personality
IP address
Uptime
TCP, UDP, ICMP ports status
Ports scripts/behavior
Droprate

Figure 1. An example case representation structure for intelligent honeypot using CBR approach.

```
(defrule windowsxp
  (or (detected_os_type "Windows XP")
      (detected_os_type "Windows XP SP1 (1)")
      (detected_os_type "Windows XP Pro SP1"))
  =>
  (assert (create windowsxp))
  (assert (personality "Microsoft Windows XP Professional SP1")))

(defrule windows2000
  (detected_os_type "Windows 2000 SP4")
  =>
  (assert (create windows2000))
  (assert (personality "Microsoft Windows 2000 Server SP3")))
```

Figure 2. An example of some rules for determining operating system fingerprint programmed in CLIPS.

VII. Conclusion and Future Works

This paper presents our research in investigating artificial intelligence approach for building intelligent honeypot. Either it is CBR or ES, both has its own way of representing knowledge and reasoning. In the domain of honeypot configurations, which is not yet a mature field, an AI approach that can reasons using past experiences seems more suitable. For future work, we would like to explore further on the practicality of the proposed approach and suggest improvements especially in its performance in real network environment.

Acknowledgment

The authors would like to thank The Honeynet Project, JColibri and CLIPS communities for providing valuable inputs.

REFERENCES

[1] Spitzner, L., 2002. *Honeypots: Tracking Hackers*. Boston, MA: Pearson Education, Inc.

[2] Chamotra, S. Deployment of a Low Interaction Honeypot in an Organizational Private Network

[3] Spitzner, L., 2001. *The Value of Honeypots, Part One: Definitions and Value of Honeypots* [online]. Available at: http://www.securityfocus.com/infocus/1492

[4] Spitzner, L. 2003a. *Definition and Value of Honeypots* [online]. Available at: http://www.tracking- hackers.com/papers/honeypots.html

[5] Leita, C., Mermoud, K. & Dacier, M., 2005. ScriptGen: An Automated Script Generation tool for Honeyd. *Proceedings of the 21st Annual Computer Security Applications Conference*, 203-214.

[6] Sadasivam, K., Samudrala, B. & T. Andrew Yang, 2005. Design of Network Security Projects using Honeypots. *Journal of Computing Sciences in Colleges*, 20 (4), 282-293.

[7] Grimes, R. A., 2005. *Honeypots for Windows*. Berkeley: Apress.

[8] HoneyBot http://www.atomicsoftwaresolutions.com/honeybot.php

[9] Dionaea http://dionaea.carnivore.it/

[10] Jiang, X. & Dongyan Xu, 2004. *BAIT-TRAP: A Catering Honeypot Framework* [online]. Available at: http://www.cs.purdue.edu/homes/jiangx/collapsar/publications/BaitTrap.pdf

[11] The Honeynet Project, 2004. *Know Your Enemy: Honeynets in Universities* [online]. Available at: http://www.honeynet.org/papers/edu/

[12] Spitzner, L., 2003b. *Dynamic Honeypots* [online]. Available at: http://www.securityfocus.com/infocus/1492

[13] Kuwatly, I., Sraj, M., Al Masri, Z. & Artail, H., 2004. *A Dynamic Honeypot Design for Intrusion Detection* [online]. Available from: http://webfea-lb.fea.aub.edu.lb/proceedings/2004/SRC-ECE-04.pdf

[14] Hecker *et al.* (2006) dynamic honeypot construction

[15] Liu, X., Peng, L. & Li, C. (2011). *The Dynamic Honeypot Design and Implementation based on Honeyd.*

[16] Chowdhary *et al.* (2004). *Towards Automatic Learning of Valid Services for Honeypots*

[17] Wagener, G. & Dulaunoy, A. (2011). *Adaptive and Self-configurable Honeypots*

[18] CLIPS: A Tool for Building Expert Systems [online]. Available at: http://clipsrules.sourceforge.net/

[19] Jess – the rule engine for the JavaPlatform [online]. Available at: http://www.jessrules.com/

[20] Finnie, G. & Sun, Z. (2003). *R^5 Model for Case based Reasoning*.

[21] JColibri GAIA – Group of Artificial Intelligence Applications [online]. Available at: http://gaia.fdi.ucm.es/research/colibri/jcolibri

[22] IUCBRF [online]. Available at: http://www.cs.indiana.edu/~sbogaert/CBR/

[23] myCBR [online]. Available at: http://mycbr-project.net/

[24] Forward chaining [online]. Available at: http://en.wikipedia.org/wiki/Forward_chaining

[25] Backward chaining [online]. Available at: http://en.wikipedia.org/wiki/Backward_chaining