Towards Automatic Learning of Valid Services for Honeypots

Vishal Chowdhary, Alok Tongaonkar, and Tzi-cker Chiueh

State University of New York, Stony Brook, NY 11794, USA

Abstract. Honeypots have emerged as an important tool in the field of *Intrusion Detection Systems*. Honeypots are decoy machines whose sole purpose is to be compromised by network attackers, in order to gain information about the attack techniques. The biggest challenge in deploying honeypots is their configuration and maintenance compounded with the fact that they either *emulate* a few services or provide the real services. The emulated services, which are usually implemented using scripts, are restricted by the responses given to the attacker. This limits the amount of information that can be gathered. The scripts are also much easier to be detected by the attacker. On the other hand, the drawback of providing real services is the greater risk associated with their use.

In this paper, we describe *service-mining*, a machine learning approach to learn and emulate behavior of real-world services. Given large enough traces of the real-service interactions and some basic information about the service, we propose a scheme whereby we can learn the semantics of its various commands and then effectively emulate the service. This service may then be deployed on a honeypot to capture attack signatures without posing a threat to the complete network.

Our initial experience in trying to emulate the popular FTP service is promising. We are able to learn the FTP service and then intelligently and consistently respond to user queries with our emulated FTP service.

1 Introduction

The growth of networked information systems has helped critical services such as commerce, banking, telecommunication, and national security. However, the spate of *hacker attacks* at many prominent commercial, military and governmental sites has highlighted the inherent risks of the networked information systems.

Intrusion Detection Systems (IDS) are used to counter this threat. Two different approaches are used to detect intrusions. *Anomaly-based* systems derive a notion of *normal* system behavior and any deviation from this profile is reported as attack. Generally, these systems are prone to a large number of false positives. So most IDS are based on *misuse detection*. The effectiveness of these systems depends on the availability of attack signatures.

Honeypots. Honeypots have emerged as an important instrument for information gathering and learning attack signature. According to [15]

A honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource.

This means that a honeypot is expected to get probed, attacked and potentially exploited. Honeypots do not fix anything. They provide us with additional, valuable information.

Honeypots are machines that are not supposed to exist in a network. Hence, all traffic from and to a honeypot is suspicious. Thus, data collected by a honeypot contains potential attack signatures. It is also of great value to give a better understanding of the attack techniques, which in turn can be used to increase overall network security. Honeypots also help in prevention of network attacks as they distract attackers from attacking real-production systems.

One important characteristic of honeypots is their level of involvement. It measures the degree an attacker can interact with the operating system and services running on a honeypot. They are usually classified as:

1. **Low-Involvement:** A low-involvement honeypot typically only provides certain emulated services [15] which are naively implemented by scripts listening to a specific port, say 80 (*HTTP*), and logging all incoming traffic. It does not have any real operating system or services. This significantly reduces the risk of the honeypot being compromised and being used in an attack. However, its role is very passive and limits the amount of useful attack information it may collect.
2. **Mid-Involvement:** In a mid-involvement honeypot the emulated services are more sophisticated. Through the higher level of interaction, more complex attacks are possible. These attacks can be logged and analyzed. The drawback of this type of honeypot is that it requires expert knowledge to code the emulated services. The popular Honeyd [9] honeypot uses a simple script as shown in Fig. 2 [1] to respond to FTP requests.
3. **High-Involvement:** A high-involvement honeypot has real operating system and services running on it. The tremendous increase in the possibility to gather important attack information is bundled with the risk of the hacker gaining complete control over the system and then using it for launching further attacks.

Table 1 [10] summarizes the features of the three types of honeypots described above.

Table 1. Features of honeypots

Feature	**Low-Invol.**	**Mid-Invol.**	**High-Invol.**
Real operating system	-	-	yes
Risk	low	mid	high
Information Gathering	connections	request	all
Knowledge to run	low	low	high
Knowledge to develop	low	high	mid-high
Maintenance time	low	low	very high

The key points of the above discussion are:

1. Developing a mid/high involvement honeypot is complex and time consuming.
2. The knowledge for developing an emulated system is very high as each protocol and service has to be understood in detail.
3. Presently, these emulated services are naively implemented as scripts or they provide responses by looking up a database of previously answered requests [9].

[1] The complete script may be found at http://www.honeyd.org/contrib/mael/ftp.sh

```
while read incmd parm1 parm2 parm3 parm4 parm5
do
    case $incmd in
        QUIT )
            echo -e "221 Goodbye." ;;
            exit 0;;
        SYST )
            echo -e "215 UNIX Type: L8" ;;
        MKD )
            echo -e "257 $parm1 new directory created." ;;
        CWD )
            echo -e "250 CWD command successful." ;;
        MLFL )
            echo -e "502 $incmd command not implemented.";;
        MAIL )
            echo -e "502 $incmd command not implemented.";;
    esac
    echo "$incmd $parm1 $parm2 $parm3 $parm4 $parm5" >> log
done
```

Fig. 1. FTP script used in Honeyd

4. The high-involvement systems should be under constant surveillance as the attacker may launch attacks via the compromised honeypot.
5. Although the low/mid involvement honeypots are easy to deploy, they limit the amount of useful information collected and are easy to detect.

In this paper, we address the problem of learning the service, to be deployed on a honeypot, through its traces. Using our emulated services the effectiveness of a low/mid involvement honeypot may be increased without affecting the risk associated with it. Our *service-mining* system is composed of four parts: **trace processor**, **flow dependence annotator**, **dependency(/rule) extractor**, and **service learner** (Fig. 2).

The trace processor collects service interactions and normalizes them into a standard format. Flow dependence annotation is the first step in refining the traces into interaction scenarios, which can be fed to the learner. It correlates the inputs/outputs of different service interactions. Next, the dependency extractor uses these them to extract dependency relations(rules) between various commands and puts them into a standard, abstract form. The service learner finally completes the learning phase by finding all possible rules that can be derived from the traces.

Paper Organization. The rest of the paper is organized as follows. We start with a formal description of our problem. Section 3 discusses the trace processor and the flow dependence annotator. Section 4 describes the dependency extractor and the service learner. In Sect. 5 we present our experimental results with FTP and briefly explain how our approach can be used to learn HTTP. Related work is discussed in Sect. 6, and Sect. 7 concludes the paper.

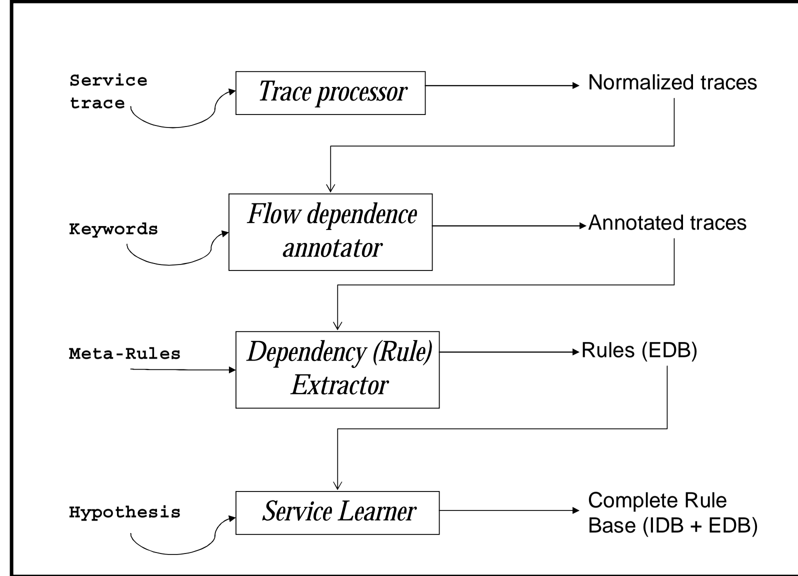

Fig. 2. Overview of service-mining

2 The Service-Mining Problem

This section states the goal of the service-miner and the prerequisites to learn a service.

Problem 1. Let I be the set of all traces of interactions with a service. Given set $L \subseteq I$, we wish to find a representation F of the service that generates exactly the traces in I. I is the set of traces which represents correct responses while interacting with the real-service and L represents the learning-set. F is the emulated service.

A service essentially is the maintenance of some state of the system. A FTP server manages files on the server machine while a database server maintains and manipulates tables in a relational database system. Each service exposes a predefined set of commands for us to interact with it. FTP has various commands like `GET`, `LS`, and `PWD` while we use SQL commands like `SELECT`, `CREATE`, and `DELETE` while interacting with a database server. Each of these commands may change the internal state of the system and/or may get information about the current state of the system. For instance, the `PUT` command in FTP and `DELETE` command in SQL change the state of the systems by adding file(s) and removing database tuple(s) respectively. On the other hand, `LS` in FTP and `SELECT` in SQL retrieve information about the state of the system in the form of list of file(s) in the current directory and the set of matching tuple(s) respectively. The service may maintain one or more states and a command may manipulate some or all of these states.

We wish to learn the semantics of the service without any prior knowledge about the service or its commands. We assume that these commands/keywords are available to our miner. Also, each of the commands take in a predefined set of arguments. Thus, an expert

user of the service initially provides the miner with a list of all possible interactions of the service and the argument list size of each interaction.

3 Trace Processor and Flow Dependence Annotator

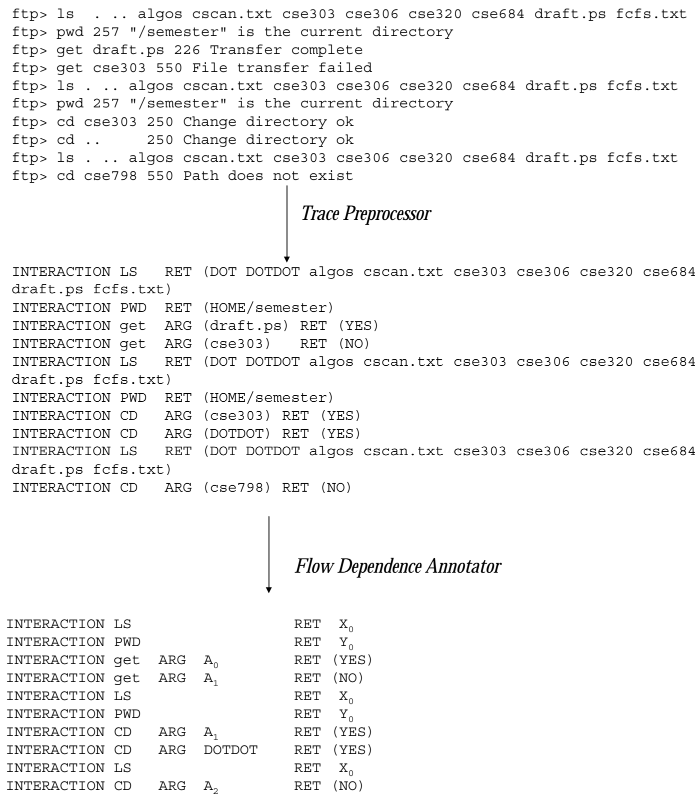

Fig. 3. Normalization and annotation of traces

The learner does not learn directly from the raw traces. Initially, the *trace processor* normalizes the traces and converts them to a standard form that serve as input for the *learner*. The advantage of normalization and converting the trace to its canonical form is that the subsequent components of the miner are abstracted from the nuances of a particular type of trace. Also we may use any tracing method or the traces made available by any user.

3.1 Canonical Traces

A canonical trace consists of *interactions*. For each command present in the trace we form an interaction. An interaction is of the following form:

$$interaction(argument_1, argument_2, ..., argument_n, response)$$

where *interaction* is the command name, got from a pre-defined set of keywords, $argument_i$ represents the input to the command and *response* [2] is the command output. All command names are treated as *constant* values. The arguments and response are *variables*. However, there may be cases where the arguments and/or response of a command are constant symbols for a particular service. For example, FTP service contains symbols `DOT` (.), `DOTDOT` (..), `YES`, and `NO` while a database server has symbols like `NULL` and `LIKE`. Any service has a small list of such symbols and an expert user can provide the miner with these symbols. In spite of being a part of the arguments/response, they are treated as constant values by the trace processor. Figure 3 shows a sample FTP trace and the result after it has been normalized by our trace-processor.

The *Flow Dependence Annotator* (FDA) examines the arguments and response of all interactions in a trace and replaces each unique value by a symbolic variable. The actual values of the symbolic variables are stored in a table T. T is used later by the dependency extractor to identify the relations between interactions. Figure 3 shows the output of the FDA when given the normalized FTP trace as input.

The output of the FDA is ready to be fed to the dependency extractor which extracts the interaction rules.

4 Dependency Extractor and Service Learner

The following section explains the workings of the dependency extractor and the service learner.

As mentioned before, every command of a service deals with the manipulation and/or query of its internal state. Our service learner attempts to learn how these states get affected by the commands.

Consider the following sequence of commands in a FTP service.

```
> ls          f1,f2,d1,d2
> cd d1       YES
> cd DOTDOT   YES
> ls          f1,f2,d1,d2
```

Now, what we observe is the pair of commands (`cd` d1, `cd` DOTDOT) has no effect on the return value of `ls`. Thus we may say that (`cd` d1, `cd` DOTDOT) together restore the internal state of the service used by `ls`. We write this as: `cd` d1, `cd` DOTDOT $\not\Rightarrow$ `ls`

Definition 1. Anti-Dependency/Rules: *Consider a sequence of commands* $D, C_1, C_2, ..., C_n, D$ *issued to a service, where the response of the first and last command,* D*, in the sequence is the same. We conclude that the sequence* $C_1, C_2, ..., C_n$ *restores the*

[2] Presently we assume no semantic knowledge about the type of response and treat them as strings. For example, the list of files returned by *LS* command is considered as a string.

state(s) of the service queried by the command D. In other words, D does NOT *depend on the set $C_1, C_2, ..., C_n$. We call such inference as an anti-dependency/rule and represent it formally as:*

$$C_1, C_2, ..., C_n \not\Rightarrow D$$

Thus, given a relation $C \not\Rightarrow D$, where both C, D represent sets, we conclude that the sequence of commands C does not have any effect on each of the commands in D. For example, given the sequence of commands:

```
> ls        f1,f2,d1,d2
> pwd       z1
> ls        f1,f2,d1,d2
> get f1    YES
> pwd       z1
> get f1    YES
```

we derive,

$$\texttt{PWD} \not\Rightarrow \texttt{GET}, \texttt{LS}$$

We employ the following heuristics to derive rules:

1. *Response:* We look at the responses of different instances of a command in a trace. If they are the same, we denote the command as D and the sequence of commands between the instances as C.
2. *Arguments and Response:* There may be correlations between the arguments and response of the same or subsequent commands. For example, the command `cd` d1 succeeds only if the response of the immediately preceding `ls` contains the directory *d1*. Direct dependencies between arguments and response are modeled as `ls` $\Rightarrow$ `cd` (`ls` affects `cd`). They are part of our initial rule-base. However, we do *not* include them for computing the closure of rules since their closure would lead to the derivation of incorrect rules. [3] Similarly when the `GET` request fails, the `HTTP` server returns an error with the name of the requested file as part of the message. We consider relationships between the arguments and response of (possibly) different commands. The expert user initially specifies N_{max}, which is the maximum number of subsequent commands above or below a command in a trace, which the service learner will look in order to derive rules.

   Definition 2. Meta-Rules *Presently, we consider both arguments and response of commands as strings. To find relationships between them, we consider Meta-Rules. These meta-rules may be specified initially for a particular service. We have kept such a provision, for we may need to consider different* types *of attributes for a particular service. In that case one may still specify the relationships and the service can be effectively learnt.*

   Intuitively, meta-rules for strings are:
   (a) *String1* is an exact match of *String2*.
   (b) *String1* is a substring of *String2* and vice-versa.
   (c) *String1* is not substring of *String2* and vice-versa.
   (d) *String1* is *String2* with all instances of a particular set of characters replaced/deleted.

[3] `ls` $\Rightarrow$ `cd`, and `cd` $\Rightarrow$ `pwd` but `ls` $\not\Rightarrow$ `pwd`.

The traces and extracted rule-base can be considered as the *extensional database* of the learner. We now proceed to the next step of deriving rules from the *extensional database* (EDB).

Hypothesis 1. Transitivity of rules: Given rules $A \not\Rightarrow B$ and $B \not\Rightarrow C$, we assume the rule $A \not\Rightarrow C$ is true. For example, given `PWD` $\not\Rightarrow$ `LS` and `LS` $\not\Rightarrow$ `GET`, we derive the rule `PWD` $\not\Rightarrow$ `GET`.

Hypothesis 2. Diminution of rules: Given rules $C_1, C_2, ..., C_n \not\Rightarrow B$ and $C_i, C_{i+1}, ..., C_n \not\Rightarrow B$, we derive the rule $C_1, C_2, ..., C_{i-1} \not\Rightarrow B$.

For example, given `PWD` $\not\Rightarrow$ `LS` and (`CD` x, `CD DOTDOT`, `PWD`) $\not\Rightarrow$ `LS`, we derive the rule (`CD` x, `CD DOTDOT`) $\not\Rightarrow$ `LS`. Note that, we may also remove any sub-sequence of commands $C_i, C_{i+1}, ..., C_j$ iff the rules $C_i, C_{i+1}, ..., C_j \not\Rightarrow C_k$ exist $\forall k : j + 1 \leq k \leq n$.

Using the above hypotheses, we generate *intensional database* (IDB) which is the complete set of rules for the service.

5 Experiments

This section presents the results of our experiment to learn the FTP [7] service from its traces. The requirements for learning the HTTP service are also discussed in brief. Finally, we discuss potential limitations of our approach.

5.1 FTP

We analyzed random traces from the *Cerberus FTP Server* [12]. Each trace contained 45 command invocations on an average.

We specified the following subset of FTP commands to our service-miner [4]:

1. `PWD`$(\phi, response)$
2. `GET`$(arg_1, response)$
3. `PUT`$(arg_1, response)$
4. `LS`$(\phi, response)$
5. `CD`$(arg_1, response)$
6. `RMDIR`$(arg_1, response)$
7. `MKDIR`$(arg_1, response)$
8. `USER`$(arg_1, response)$
9. `PASS`$(arg_1, response)$

FTP contains the following keywords `DOT, DOTDOT, YES, NO`, which were specified initially to the learner. The meta-rules are same as that for strings as specified in Sect. 4. For each trace, our service-miner learnt about 16 rules using our heuristic of considering the response alone. After employing the heuristics and computing the closure, the total rule base, for each trace, contained on an average 80 rules.

Using the IDB (Sect. 4), the table T (Sect. 3) containing actual values of the symbolic variables and the current state, we were able to respond to subsequent FTP requests intelligently and emulate the fake responses consistently.

[4] ϕ indicates method with no inputs.

5.2 HTTP

HTTP [8] is the primary request/response protocol between clients and servers on the web. After establishing a TCP connection, the client sends a request string requesting a particular file from the server. The server would then respond with the file (or an error message).

To learn the HTTP service, we initially specify the set of all interactions to our service-miner. Thus the definitions for `OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT` are specified by the expert user. The keywords for HTTP would be the status definition codes (`1xx, 2xx, ..., 5xx`). The meta-rules would be the ones specified for strings.

Given large enough traces, we could easily learn that the `HEAD` and `GET` responses are related (one being a substring of the other), the error codes follow some pattern (access to a directory unauthorized), and there does not exist any further dependencies between any two commands. The emulated HTTP service can then be utilized to intelligently respond to client requests.

5.3 Limitations

The current approach relies on accurate command definitions, keywords and meta-rules to be specified by the expert user. Also, the learning is limited by the rules that may be derived using the initial traces. We hope to build a meta-rule database that intelligently adapts to the service.

6 Related Work

Program learning approaches fall into two broad categories: static analysis and dynamic analysis. Static analysis techniques extract program models from source code or binaries. A number of static analysis techniques have been used to generate program models. [17] extract PDA models from *C* source code for intrusion detection.

We follow the dynamic analysis approach. Dynamic approaches ([6], [3], [5], [11]) construct program models from observed program behavior. The above approaches extract program behavior in terms of system calls. [2] use dynamic analysis to discover formal specification of the protocols that the code must obey when interacting with an application program interface or abstract data type.

Presently, there are both free and commercial honeypots available. Back Officer Friendly [4], Deception Toolkit (DTK) [16] and Specter [14] are honeypots that emulate services using scripts. Symantec Decoy Server [13] and Honeynets [1] are high-involvement honeypots that use real operating systems and services. As mentioned before, the risks involved is high in these honeypots.

7 Conclusion and Future Work

This paper addresses an important problem of learning program behavior. Our emulated programs can be deployed on low/mid involvement honeypots increasing their signature capturing abilities with lower risk levels. We have provided an algorithm for service-mining. While the tests are preliminary and we have yet to deploy this on a real honeypot,

the initial experience with the FTP and HTTP services is promising. We plan to consider more services and integrate our emulated services with Honeyd and capture interesting attack patterns.

References

1. Honeynet Research Alliance. http://www.honeynet.org.
2. Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Symposium on Principles of Programming Languages*, pages 4–16, 2002.
3. Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.
4. Back Officer Friendly. www.nfr.com/resource/backOfficer.php.
5. Wenke Lee and Salvatore Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, 1998.
6. Wenke Lee, Salvatore Stolfo, and Patrick Chan. Learning patterns from unix process execution traces for intrusion detection. In *Proceedings of the AAAI97 workshop on AI methods in Fraud and risk management*, 1997.
7. File Transfer Protocol. www.faqs.org/rfcs/rfc959.html.
8. Hypertext Transfer Protocol. www.w3.org/Protocols/rfc2616/rfc2616.html.
9. Niels Provos. Honeyd - a virtualhoneypot daemon. In *10th DFN-CERT Workshop*, Hamburg, Germany, 2003.
10. Baumann R. and Plattner C. Honeypots. Master's thesis, Swiss Federal Institute of Technology, Zurich, 2002.
11. R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, pages 144–155, 2001.
12. Cerberus FTP Server. www.cerberusftp.com.
13. Symantec Decoy Server. http://www.enterprisesecurity.symantec.com.
14. Specter. www.specter.com.
15. Lance Spitzner. *Honeypots - Tracking Hackers*. Addison-Wesley, 2002.
16. Deception Toolkit. http://all.net/dtk/dtk.html.
17. D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, pages 156–169, 2001.