A Practical Guide to Honeypots

Eric Peter, epeter(at)wustl(dot)edu and **Todd Schiller**, tschiller(at)acm(dot)org (A project report written under the guidance of Prof. Raj Jain)

Download 

Abstract

This paper is composed of two parts: a survey of honeypot technology and a case study describing a low-interaction honeypot implemented in Java. The survey provides a brief overview of honeypot concepts and references to more detailed works. The implementation case study addresses many of the decisions required when designing and implementing a low-interaction honeypot.

Keywords

Honeypot, network security, low-interaction, honeypot implementation, honeypot trends, honeyd, specter, honeyBot, case study

Table of Contents

---

1 Introduction

In this section we describe network intrusion detection systems, the traditional approach to network security. We then introduce and provide a brief history of honeypots. The section concludes with a discussion of the general advantages and disadvantages of honeypots.

1.1 Network Intrusion Detection Systems

The goal of an Intrusion Detection System (IDS) is to "identify, preferably in real time, unauthorized use, misuse, and abuse of computer systems by both system insiders and external penetrators" [Mukherjee94]. An IDS is used as an alternative (or a complement) to building a shield around the network. The shielding approach is deficient in several ways, including failure to prevent attacks from insiders. Mukherjee et al. provide an overview of the IDS problem [Mukherjee94].

Despite the groundwork being laid for detection systems, it wasn't until Paxson's work in 1998 that methods for building real-time detection systems became publically available. The system converts a stream of packets into a series of high-level network events that can be analyzed according to system security policy [Paxson99]. Since 1999, this work has been extended to incorporate advanced machine learning techniques [Verwoerd02] and better detect threats such as denial-of-service attacks [Hussain03].

However, while IDS technology is progressing, methods to circumvent IDSs are becoming more prevalent [Ptacek98]. For example, Wagner and Soto develop a class of mimicry attacks which mimic the original behavior of the application [Wagner02]. In light of these attacks as well as the growing prevalence of encrypted communication, alternatives such as honeypots have become more popular.

1.2 Honeypots

The exact definition of a honeypot is contentious, however most definitions are some form of the following:

A honeypot is an "an information system resource whose value lies in unauthorized or illicit use of that resources"(from the www.securityfocus.com forum)

A more practical, but more limiting, definition is given by pcmag.com:

"A server that is configured to detect an intruder by mirroring a real production system. It appears as an ordinary server doing work, but all the data and transactions are phony. Located either in or outside the firewall, the honeypot is used to learn about an intruder's techniques as well as determine vulnerabilities in the real system" [pcmag09].

In practice, honeypots are computers which masquerade as unprotected. The honeypot records all actions and interactions with users. Since honeypots don't provide any legitimate services, all activity is unauthorized (and possibly malicious). Talabis presents honeypots as being analogous to the use of wet cement for detecting human intruders [Talabis07b].

1.3 Honeypot History

The first publically available honeypot was Fred Cohen's Deception ToolKit in 1998 which was "intended to make it appear to attackers as if the system running DTK [had] a large number of widely known vulnerabilities" [Cohen98]. More honeypots became both publically and commercially available throughout the late nineties. As worms began to proliferate beginning in 2000, honeypots proved imperative in capturing and analyzing worms. In 2004, virtual honeypots were introduced which allow multiple honeypots to run on a single server [Provos04].

A detailed history of honeypots can be found in [Spitzner02] and [Talabis07a].

1.4 Types of Honeypots

There are two broad categories of honeypots available today, high-interaction and low-interaction. These categories are defined based on the services, or interaction level, provided by the honeypot to potential hackers [Spitzner02]. **High-interaction honeypots** let the hacker interact with the system as they would any regular operating system, with the goal of capturing the maximum amount of information on the attacker's techniques. Any command or application an end-user would expect to be installed is available and generally, there is little to no restriction placed on what the hacker can do once he/she comprises the system. On the contrary, **low-interaction honeypots** present the hacker emulated services with a limited subset of the functionality they would expect from a server, with the intent of detecting sources of unauthorized activity [Provos04]. For example, the HTTP service on a low-interaction honeypot would only support the commands needed to indentify that a known exploit is being attempted. Some authors classify a third category, medium-interaction honeypots, as providing expanded interaction from low-interaction honeypots but less than high-interaction systems. A **medium-interaction honeypot** might more fully implement the HTTP protocol to emulate a well-known vendor's implementation, such as Apache. However, there are no implementations of a medium-interaction honeypots and for the purposes of this paper, the definition of low-interaction honeypots captures the functionality of medium-interaction honeypots in that they only provide partial implementation of services and do not allow typical, full interaction with the system as high-interaction honeypots.

1.5 General Honeypot Advantages and Disadvantages

Honeypots provide several advantages over other security solutions, including network intrusion detection systems:

- Fewer false positives since no legitimate traffic uses honeypot
- Collect smaller, higher-value, datasets since they only log illegitimate activity
- Work in encrypted environments
- Do not require known attack signatures, unlike IDS [Provos07]

Honeypots are not perfect, though:

- Can be used by attacker to attack other systems [hnet08]
- Only monitor interactions made directly with the honeypot - the honeypot cannot detect attacks against other systems
- Can potentially be detected by the attacker

Tradition security solutions, such as intrusion detection systems, may not be enough in light of more complicated attacks. Honeypots provide a mechanism for detecting novel attack vectors, even in encrypted environments. Advances such as virtualization have made honeypots even more effective. Honeypots have drawbacks, though, so it is important to understand how honeypots operate in order to maximize their effectiveness.

2 Recent Trends and Advances

In this section, we discuss the trend towards grouping honeypots into honeynets or honeyfarms. Shadow Honeypots and distributed honeypots, both cutting edge technologies, are then introduced.

2.1 Honeynets and Honeyfarms

Honeynets and honeyfarms are the names given to groups of honeypots. Honeyfarms tend to be more centralized. Grouping honeypots provide many synergies that help to mitigate many of the deficiencies of traditional honeypots. For instance, honeypots often restrict outbound traffic in order to avoid attacking non-honeypot nodes. However, this restriction allows honeypots to be identified by an attacker. He et al. use honeyfarms as redirection points for outbound traffic from each individual honeypot. These redirection nodes also behave like real victims [He04]. Figure 1 shows the redirection of outbound traffic from a honeypot to another node in the honeyfarm.

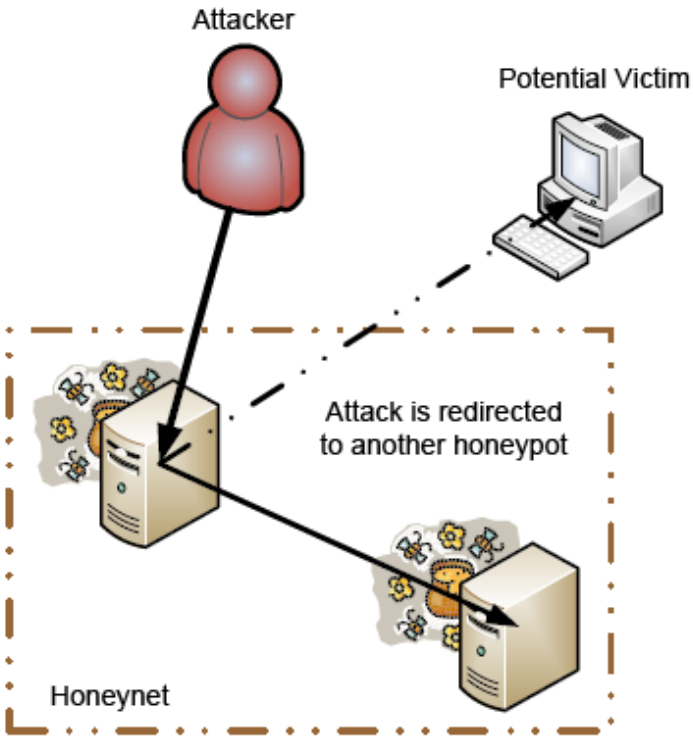

Figure 1. Redirecting an outbound attack in a honeynet

2.2 Shadow Honeypots

Shadow honeypots are combination of honeypots and anomaly detection systems (ADS), which are another alternative to rule-based intrusion detection systems [Anagnostakis05]. A comparison of ADS with other detection systems can be found in [Kemmerer02].

Shadow honeypots first segment anomalous traffic from regular traffic. The anomalous traffic is sent to a shadow honeypot which is an instance of a legitimate service as shown in Figure 2. If an attack is detected by the shadow honeypot, any changes in state in the honeypot are discarded. If not, the transaction and changes are correctly handled. While shadow honeypots require more overhead, they are advantageous in that they can detect attacks contingent upon the state of the service. Implementation studies are offered in [Anagnostakis05]

and [Euro07].

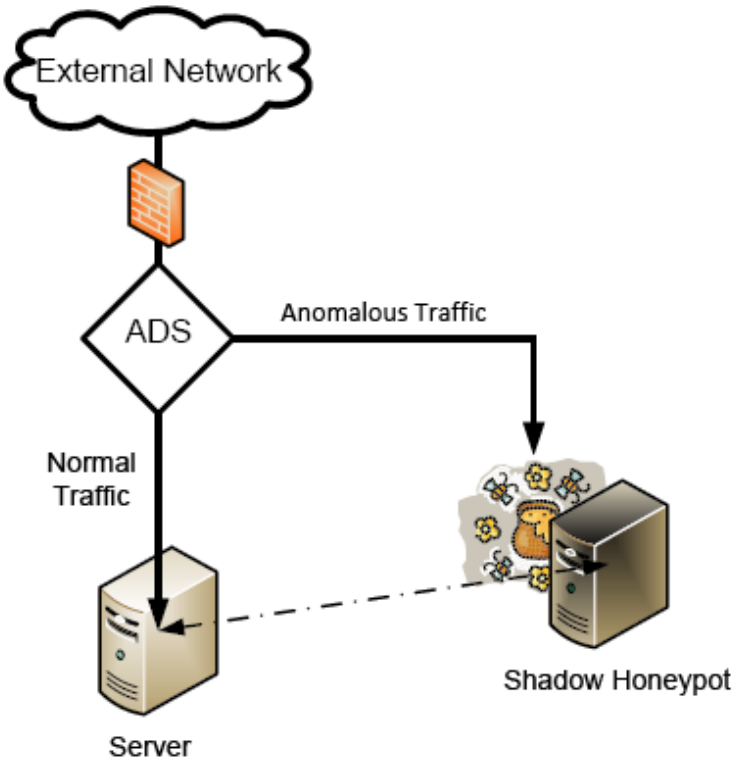

Figure 2. Segmenting traffic in a shadow honeypot system

2.3 Distributed Honeypots

One disadvantage of honeypots is that must take up a large portion of the address space in order to be efficient and useful (since attackers and malware must target the honeypots). Yang et al. provide a distributed framework for grid computing in which legitimate hosts redirect suspicious users to a single honeypot [Yang04]. An alternative is used by Honey@home in which each client is responsible for a single unused IP address. The client traffic is redirected anonymously through the Tor network to a collection of central honeypots [Antonatos07].

Honeyfarms, honeynets, and distributed honeypots all address the need to monitor a large set of network addresses in order for a honeypot to be effective. As discussed in Section 2.1, grouping honeypots can also add functionality to honeypots by allowing for operations such as simulated outbound traffic. Honeynets, shadow honeypots, and distributed honeynets are just a few of the advances occurring in the field of honeypots. We encourage you to explore journals and online to read about the latest advances.

---

3 Existing Honeypot Products

In this section, we provide a very brief survey of the Honeyd, HoneyBOT, and Specter honeypots. For each, we describe what differentiates the solution and provide reference information. We then offer advice for selecting amongst the solutions.

3.1 Honeyd

Honeyd is a honeypot for linux/unix developed by security researcher Niels Provos. Honeyd was ground-

breaking in that it could create multiple virtual hosts on the network (as opposed to just using a single physical host). The honeypot can emulate various operating systems (which differ in how they respond to certain messages) and services. Since Honeyd emulates operating systems at the TCP/IP stack level, it can fool even sophistic network analysis tools such as nmap. Upon attack, Honeyd can passively attempt to identify the remote host. The Honeyd project is located at http://www.honeyd.org/

3.2 HoneyBOT

HoneyBOT is a Windows medium-interaction honeypot by Atomic Software Solutions (http://www.atomicsoftwaresolutions.com/honeybot.php). It originally began as an attempt to detect by the Code Red and Nimda worms in 2001 and has been released for free public use since 2005. HoneyBOT allows attackers to upload files to a quarantined area in order to detect trojans and rootkits. HoneyBOT's user interface is shown in Figure 3.

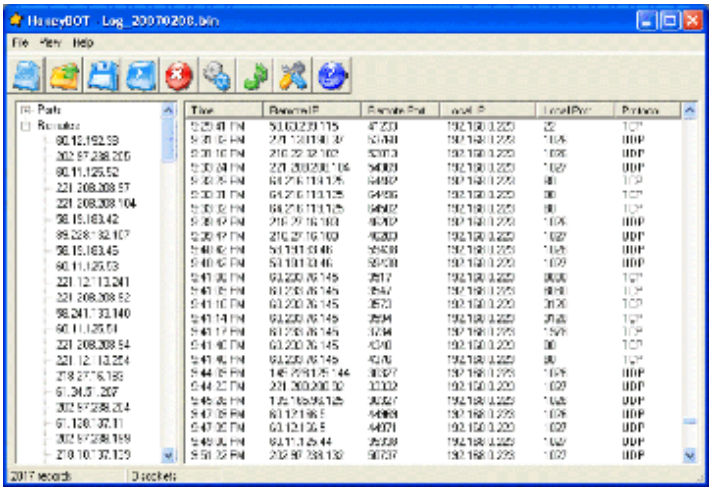

Figure 3. The main HoneyBOT user interface. Taken from http://www.atomicsoftwaresolutions.com/screenshot.php.

3.3 Specter

Specter's authors describes Specter as a "honeypot-based intrusion detection system". However, the product is primarily a honeypot designed to lure attackers away from production systems and collect evidence against the attackers. Specter has a few interesting features not found in other solutions:

- Specter makes decoy data available for attackers to access and download. These data files leave marks on the attacker's computer as evidence
- Specter can emulate machines in different states: a badly configured system, a secured system, a failing system (with hardware or software failures), or an unpredictable system.
- Specter actively attempts to collect information about each attacker

Figure 4 shows Specter's master control center. Specter can be found online at http://www.specter.com/

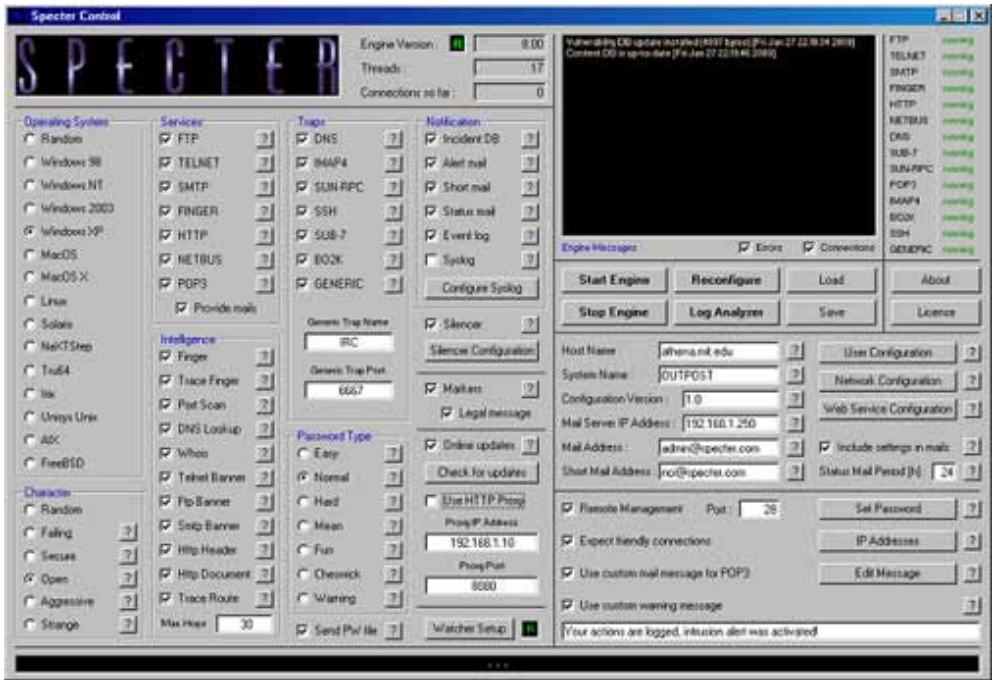

Figure 4. The Specter control center. Taken from http://www.specter.ch/details50.htm.

3.4 Comparison

HoneyBOT is a great way to start exploring the world of honeypots. Though it lacks the functionality of honeyd and Specter (and is closed-sourced) it allows users to quickly run a honeypot. As an open-source honeypot, honeyd is fully extensible. Though it has many complex features, such as virtual network topology design, the core honeypot technology is easy to use. Specter is closed-source and not free. But, as a commercial product, a lot of attention has been paid to constructing a graphical user-interface and help system. Moreover, it includes many unique (and patent-pending technologies). For example, Specter can leave verifiable markers on attacker's computers. Table 1 provides a brief comparison of the honeypots.

Honeypot	Interaction Level	OS Emulation	Price	Open/Closed Source
honeyd	Low	Yes	Free	Open
HoneyBOT	Low	No	Free	Closed
Specter	High	Yes	$599 - Light Version $899 - Full Version	Closed

Table 1. Comparison of honeyd, HoneyBOT, and Specter

---

4 HoneyRJ: Low-Interaction Honeypot Implementation Case Study

This application, HoneyRJ, is an implementation of a low-interaction honeypot. As defined above, a low-interaction honeypot serves a number of limited functionality protocols with the intent of capturing the source of traffic coming to the honeypot. A honeypot is located on an IP address that is used solely for the purpose of the honeypot and not for any legitimate services; any connections to the software are presumed to be malicious and are logged for later review.

HoneyRJ was designed to be extremely simple and easily extendable. Our design decisions reflect the desire to create a simple application that demonstrates the concept of a low-interaction honeypot and allows anyone with minimal technical knowledge to extend the application to include their desired protocols.

4.1 Implementation Features

HoneyRJ supports the following basic features:

- **Support for multiple protocols** - The application supports the addition of any protocol a user is willing to program by implementing the provided Java Interface. Any class implementing the interface can be added to the HoneyRJ application and HoneyRJ will interact with clients according to the logic defined in that class. See section 4.4 for details on implementing protocols.
- **No limit to number of client connections** - HoneyRJ has a multi-threaded design so that it can listen for connections and talk with any number of clients simultaneously.
- **Logging** - HoneyRJ creates a log file for each connection and logs all sent and received packets.
- **Graphical Interface** - HoneyRJ provides a simple GUI to allow the user to control the application.
- **Configurable** - The application can be configured at compile time to adjust several options.

The following denial of service (DOS) attack prevention features are implemented in HoneyRJ:

- **Connection timeout**
  - A hacker could attempt to launch a DOS attack on the honeypot by opening a large number of connections and leaving them in an idle state. This could prevent an administrator or another hacker from opening a connection to the machine running HoneyRJ because the operating system will have exhausted its network resources.
  - Each connection to HoneyRJ will be closed after configured timeout period (by default, 2 minutes). If the connection is idle for the timeout period, HoneyRJ will forcefully close that connection. If a connection never becomes idle, HoneyRJ will forcefully close the connection after it is connected for the timeout period.
- **Waiting period for multiple connections**
  - A hacker could attempt to launch a DOS attack on the honeypot by rapidly opening connections, potentially utilizing a large amount of system resources. This could prevent the honeypot from capturing traffic from other hackers or starting new connections.
  - HoneyRJ forces a configured period of time (by default, 5 seconds) between simultaneous connections on a protocol. During this period, the hacker cannot create new connections to the protocol.

4.2 Application Design

This section provides an overview the application design process, including pre-design decisions, relevant design decisions and documentation on the application internals. We outline our choice of developmental language and environment, and then provide our thinking on key design decisions including the multithreaded design, logging format and security implications. Finally, we document the internal flow of the application and provide a section on how to write and install an additional protocol for HoneyRJ.

4.2.1 Choice of Development Language

We chose Java for the development language for a number of reasons. First, we are very familiar with Java through our previous coursework and outside experience, allowing us to focus our time on the design and development, not the learning of a language. Secondly, Java provides a stable and easy-to-use high-level Sockets implementation, allowing us to not have to learn low-level socket programming, further allowing us to concentrate on design and development. Finally, Java provides an excellent thread library, which eased our implementation of HoneyRJ as a multi-threaded application.

4.2.2 Choice of Integrated Development Environment (IDE)

an IDE is an application that provides software developers with an environment that eases tasks related to software programming. We chose Eclipse as the IDE in which we developed the project. Eclipse is a free, open source product and is supported by the Washington University CEC. It provides all the features one would expect in a modern IDE: code completion, refactoring and package management. Eclipse also has built in support for JavaDoc, which allowed easy generation of source code documentation. We are intimately familiar with the Eclipse environment through coursework and outside experience, allowing us to start programming without the associated learning curve of an unfamiliar IDE.

4.2.3 Design Decisions

We have outlined our thinking behind the design decisions that occurred while developing HoneyRJ. The result of each decision is outlined below, with detail provided as to how we reached that decision. Our decisions follow our desire to create a simple, yet extendable, low-interaction honeypot robust enough to run multiple protocols and talk with multiple clients simultaneously.

4.2.3.1 Multi-threaded to support multiple connections

HoneyRJ can listen on multiple protocols and can talk to multiple clients on each protocol at once. We decided to design HoneyRJ to support multiple connections because otherwise the application would be severely limited in terms of its usefulness as a honeypot: the application would only be able to log one hacker's connection at a time. With only one available connection, you would not be able to run multiple protocols or see multiple connections from one hacker. This would be a severe limitation on the usefulness of the data collected by HoneyRJ and thus we decided to implement HoneyRJ as a multi-threaded application.

Our multi-threaded design is loosely based on the design in the "Supporting Multiple Clients" section of the Java Sockets tutorial [JavaTutorial2008]. Each protocol running in HoneyRJ has a thread listening for connections on a socket. As shown in figure 5, when a thread receives a connection, a worker thread is launched to speak with the connected client, while the main thread continues to listen for connections. This design allows the application to listen on multiple ports simultaneously, while talking with multiple clients.

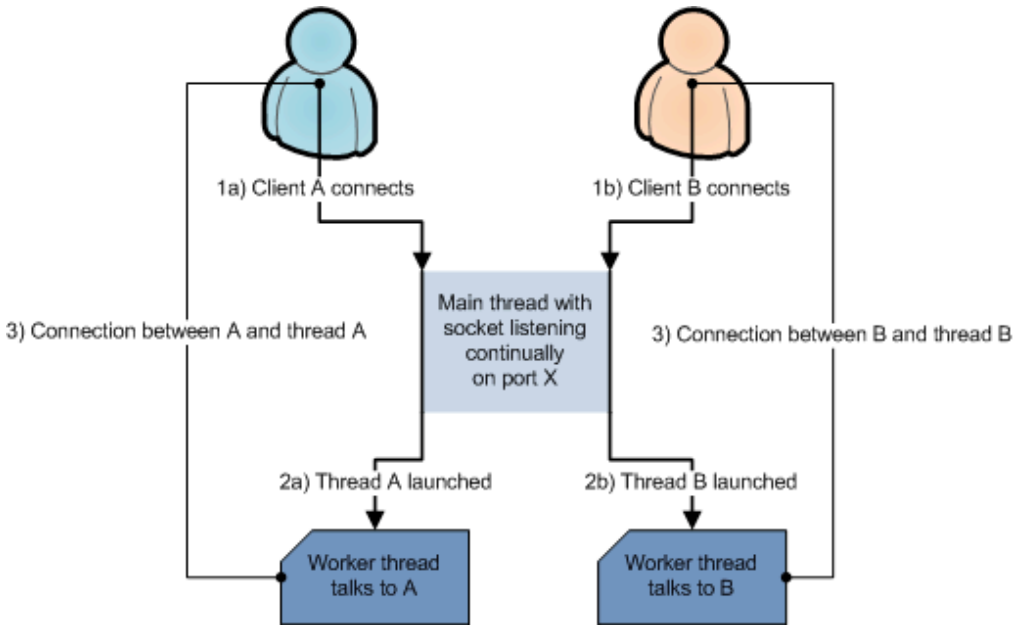

Figure 5. Two clients connecting on the same port

4.2.3.2 Logging to plain text files

HoneyRJ stores log files as text documents in a local directory, updating them as the connection progresses. We decided to store logs as plain text documents to allow a user to easily read them and to allow parsing by third-party utilities. Alternatively, we could have stored the log files as serialized Java objects; however this would require a viewer application and would prevent easy parsing in the future. We chose to continually update the log file as a connection progresses to allow a user to monitor active connections by viewing the log file. This provides the user with more flexibility than writing log files at the end of connections and protects from log data going missing upon an application crash. Finally, if the user only wishes to view completed log files, they can choose to only open text files that represent closed connections.

The format of log files is defined in section 5.2.1 of the user manual and is loosely based on the ipfw log file format [OpenDoor07]. We chose to store the timestamp, sending and receiving IP addresses and ports and the contents of the packet because this information is directly relevant to identifying where the clients are located and their actions once connected.

4.2.3.3 Easy creation of additional protocols

HoneyRJ provides the ability to create a new protocol by implementing a provided Java Interface. We designed the application this way to allow a user to write additional protocols beyond the sample protocols provided with HoneyRJ. Non-standard protocols (for example, a protocol specific to a company's product) can be written for HoneyRJ using this architecture. In addition to providing a high degree of customization to the end user, this design decision made it easier to implement the sample protocols. The process of writing a protocol is described in more detail in section 4.4 Implementing Additional Protocols.

4.2.3.4 Support only for string based protocols

HoneyRJ only supports string-based protocols and does not support the transmission of binary data. We implemented HoneyRJ this way primarily for simplicity; however, there are security implications related to allowing users to upload binary files. For example, a hacker could upload a binary file with a virus and then execute it through a buffer overflow attack present in the operating system running HoneyRJ. Many protocols are text-based and thus HoneyRJ is able to support most protocols a user would want to implement.

4.2.3.5 DOS attack prevention

HoneyRJ was designed with a connection timeout and waiting period between connections to a protocol. This design prevents denial of service attacks from a malicious user. We included these safeguards because the target audience that will be connecting to HoneyRJ is not trusted and in fact, are solely making malicious connections. Without these safeguards in place, a hacker might be able launch a DOS attack against the machine running HoneyRJ. Any connection left open to HoneyRJ will automatically be disconnected after the configured timeout (by default, 2 minutes). This prevents a hacker from leaving thousands of connections in an open state and thereby preventing other users or the administrator from connecting to the machine running HoneyRJ. In addition, once a protocol accepts a connection, it will wait a configured period of time (by default, 5 seconds) before accepting another connection on that protocol. This prevents a user from opening a large number of connections in a short period of time.

4.3 Application Internals

This section describes the flow of events inside HoneyRJ when the application is launched and when a module is added and started. This section references the non-GUI classes only, as the GUI only provides wrapper functionality to these classes. Included in the appendix is the JavaDoc for HoneyRJ, which provides a description of every method and variable used in the application. If you desire to make significant changes to HoneyRJ, we recommend reading the JavaDoc before starting. The HoneyRJ application consists of 2 main

classes, **HoneyRJ** and **LIModule**, and two helper classes, **LIModuleThread** and **LIProtocol**.

HoneyRJ is the main application class and controls multiple LIModule classes, which provide connection support to implemented protocols. Each LIModule contains a class implementing the LIProtocol interface to provide communication logic with connected clients. A LIModuleThread is launched by a LIModule upon client connection to communicate with the client.

4.3.1 HoneyRJ Launch and Initialization

This section details the initialization of the main application class, HoneyRJ. The HoneyRJ class contains one or more modules and provides services relating to managing these modules. An overview of the launch and initialization is shown in figure 6. Upon application launch, the HoneyRJ class constructor is called. The constructor creates a HashMap structure to store the contained LIModules. The HashMap maps a port number to a LIModule, allowing the application to ensure only one module is loaded for each port. Once the HashMap is initialized, the logging directory is created and a reference to this directory is saved as a member variable for later passing to added LIModules. At this point, HoneyRJ is ready to have modules added. An instance of a LIModule is initialized *(see section 4.3.2 for details on the initialization of a module)* and is passed to the RegisterService() method of the HoneyRJ class, which adds it to the HashMap, ensuring no other modules are defined for its port. Once added to the HashMap, the module's registerParent() method is called, giving it access to the logging directory provided by HoneyRJ. Additional modules are then added by repeating this process. At this point, HoneyRJ is waiting for the user to start the added modules. Once the user starts the modules, the application will be listening for connections.

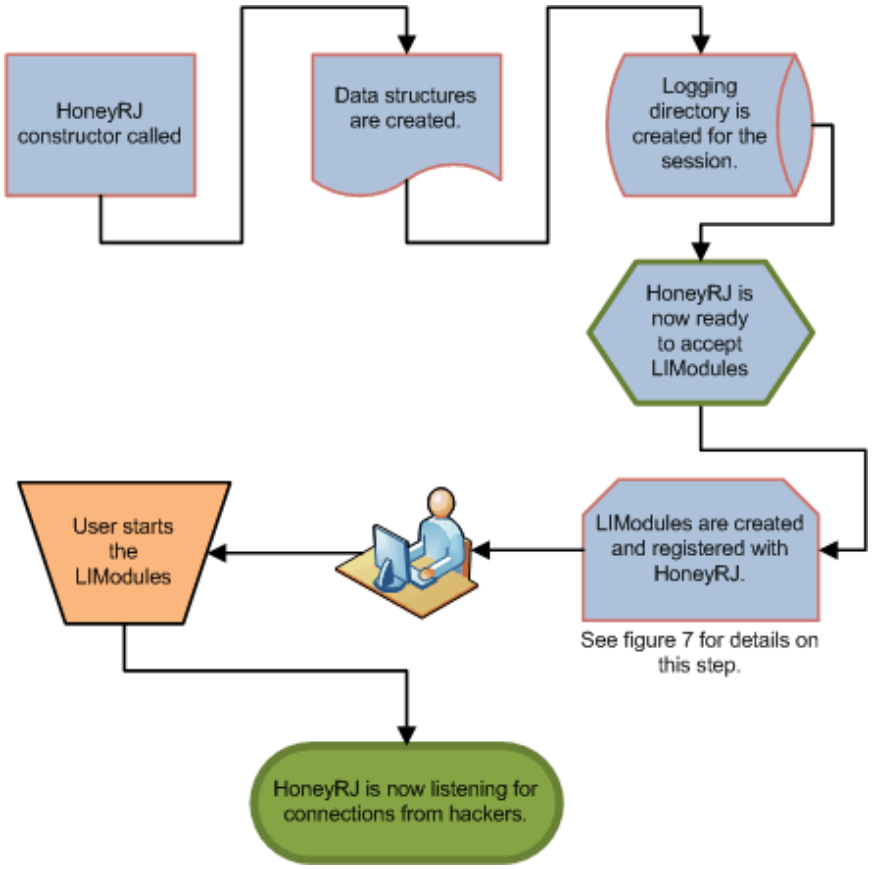

Figure 6. HoneyRJ application launch flow

4.3.2 LIModule Initialization

This section details the process of initializing a LIModule and the steps that occur when a LIModule is started.

A LIModule handles the communication and logging relating to one protocol. An overview of the process is shown in figure 7. To create an LIModule, the LIModule constructor is called with an initialized class implementing the LIProtocol interface (see section 4.3.3 for details on the LIProtocol interface). In the constructor, the given LIProtocol class is stored as a member variable. At this point, the LIModule waits for a HoneyRJ class to register itself with the module by calling the registerParent() method. Once registered, the LIModule stores a reference to the parent for later access to the parent's logging directory information. The module is now ready to be started by a user.

Once started, the module launches itself into a thread and creates a ServerSocket [JavaDocumentation2004] listening on the port specified by the contained LIProtocol. When a client connects to the port, the LIModule launches a LIModuleThread worker thread with the connected socket. The LIModuleThread communicates with the hacker according to the LIProtocol, while the LIModule continues to listen for new connections.

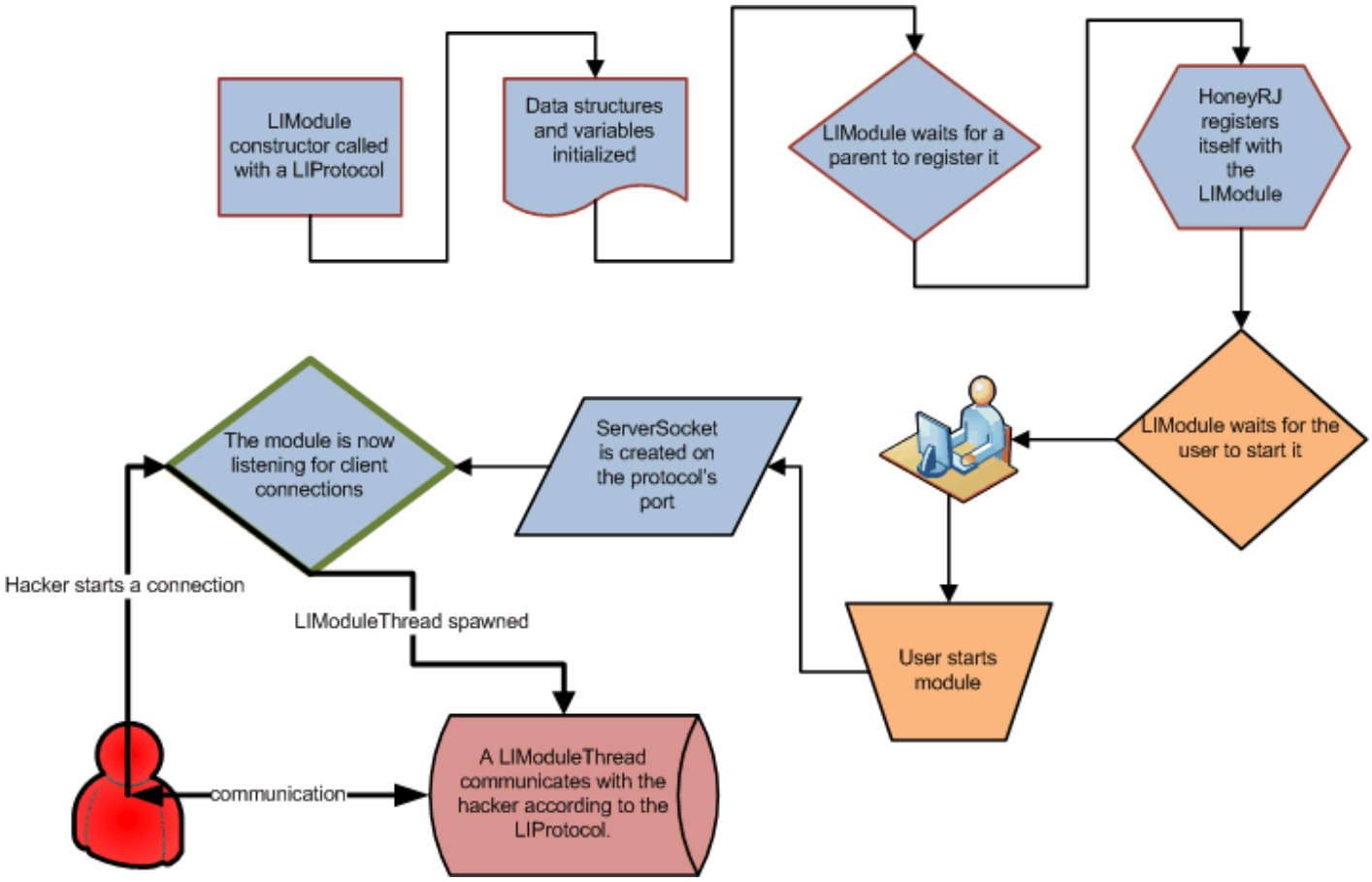

Figure 7. The creation, initialization and startup of a LIModule

4.3.3 LIProtocol interface overview

The LIProtocol defines five methods that must be implemented by the protocol's class. Of these five methods, the processInput() method does the majority of the work, while the other four methods provide information about the protocol. When a LIModuleThread is launched to handle a client connection, it creates an instance of the class implementing the LIProtocol interface. The process of receiving and sending messages is outlined in figure 8, using the FTP protocol as an example. Each packet received from the client on the socket is converted into a String object and passed as a parameter to the processInput() method. The processInput() method is then expected to process that String and return its response to the client as a Vector of String objects. Each String in the returned Vector is sent to the client as a separate line. If your protocol only returns one String, a helper method, LIHelper.vectorFromString(), is provided to create a Vector object from one String.

The other methods of the LIProtocol interface are defined as follows:

- whoTalksFirst() - The return value of this method lets the LIModuleThread know if the client or server sends the first message once a connection is established. In HoneyRJ terminology, this is referred to as who "talks first." It is defined by the TALK_FIRST enum and has two values: SVR_FIRST (if the server sends the first message) and CLIENT_FIRST (if the client sends the first message).
    - A protocol that is defined as having the server talk first will have its first message requested by a call to processInput() with a null object as the parameter. The implementation should recognize this and return the first message.
- getPort() - The integer return value of this method lets the LIModule know what port the protocol listens on.
- toString() - The String return value of this method is used to name the log files and identify the protocol in the GUI. It should return a String specifying the name of the protocol, for example, "FTP."
- isConnectionOver() - The Boolean return value of this method indicates to the LIModuleThread if the protocol believes the connection is over, that is, no more messages should be sent or received. The return value of this method is checked after the String objects returned by processInput() are sent to the client. The method should return a Boolean true if the connection is over or a Boolean false if the protocol is still expecting to send or receive messages.

For the complete specification of the methods (including Java return types and parameter types), please view the JavaDoc included in the appendix.

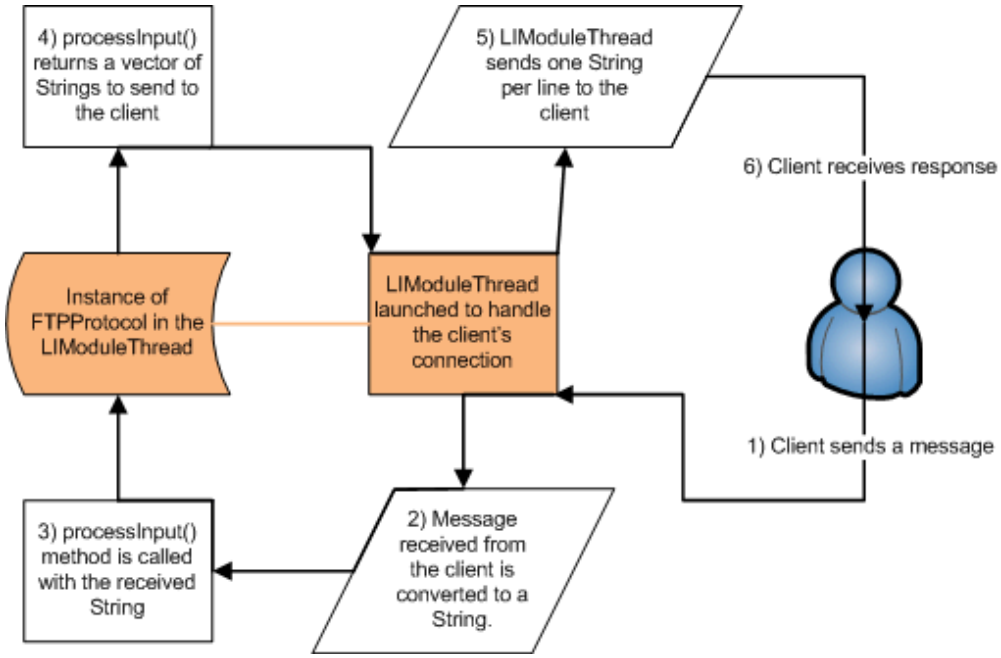

Figure 8. The FTP protocol's communication with a client through the processInput() method

In this section, we reviewed the classes that make up the HoneyRJ application. HoneyRJ is the main class which contains LIModule classes, each of which host one protocol. Protocols are defined by implementing the LIProtocol interface. When a client connects, an LIModuleThread is spawned to talk to that client.

4.4 Implementing Additional Protocols

HoneyRJ allows a user to write additional protocols and "plug" those protocols into HoneyRJ. This section outlines the steps required to implement a protocol. Throughout the section, all examples are provided in the context of the FTP protocol that we developed. The process for creating a new protocol begins with several key

design decisions. You will then create a class implementing the 5 methods in the LIProtocol interface. You then add a reference to the created class in the main application method through simple modification to the HoneyRJMain.java file.

The process of writing a protocol requires knowledge of Java programming: at minimum, an understanding of common Java data structures and object oriented programming. In this section, we assume the user is using the Eclipse integrated development environment. In addition, we assume you have intimate knowledge of the protocol you will be implementing.

4.4.1 Key Protocol Design Decisions

Before you begin programming, you need to make several key decisions about your protocol:

- **What is the name of your protocol?** Decide on a short name (preferably under 5 characters) for the protocol that will be used for display in the GUI and to name the log files. For example, "FTP" is the name of the FTP protocol.
- **Once the connection is established, does the server or client talk first?** FTP is an example of a protocol in which the server talks first: once the connection is established, the client waits for the server to send a "220 Service Ready" message. IRC is an example of a protocol in which the client talks first: once the connection is established, the server waits for the client to send a "NICK username" message.
- **On what port does your protocol operate?** Most common protocols have a port defined in their defining RFC document. If you are implementing a custom protocol, this is a decision you must make on your own, however, we recommend **not** re-using a well known port for a custom protocol. A list of well known ports is available from the IANA at http://www.iana.org/assignments/port-numbers. FTP runs on port 21.

4.4.2 Implementation

Armed with the answers to the design decisions, you can begin programming your protocol. First, import the Eclipse project provided as part of the source code into Eclipse following the instructions available in the Eclipse documentation [Eclipse08]. The next step is to create a public class that implements the LIModule interface. We recommend naming the class as [Name]Protocol, replacing [Name] with the name you decided in the key decisions section, and storing the class in the **src/protocol** package inside the Eclipse project.

- **public class** FtpProtocol **implements** LIProtocol
  - The class implementing the FTP protocol is called "FtpProtocol."

Once you have the class created, allow Eclipse to generate skeleton methods that implement the interface. You will need to create a toString() method using the "Override/Implement methods" function in Eclipse. The toString(), getPort() and whoTalksFirst() methods are simple to implement and represent the answers to the key decision questions.

- **public** TALK_FIRST whoTalksFirst(){**return** TALK_FIRST.SVR_FIRST; }
  - The FTP protocol requires the server to send the first message.
- **public** String toString(){**return** "FTP"; } return
  - We decided to name the FTP protocol FTP.
- **public int** getPort(){**return** 21; } return
  - The FTP protocol listens on port 21.

The remaining two methods, isConnectionOver() and processInput(), are more challenging, as a proper implementation requires remembering the state of the connected client. The FTP protocol implementation uses a member variable **connectionState** to store the state of the connection and implements a switch() statement on

this variable in the processInput() method to determine if the proper input was received and the proper response to send. isConnectionOver() is implemented by checking if the connectionState variable is equal to the constant representing the closed state.

The processInput() method returns a Vector of String objects in response to a String. If you require only one String in response to a message, the static helper method LIHelper.*vectorFromString*() is provided to save you the time of encapsulating each response String as a Vector. The static method returns a Vector with the given String as the only member.

- **public** Vector<String> processInput(String msg){ /* see source code for implementation */ }
- **public boolean** isConnectionOver(){**return** connectionState == *KILLED*; }

4.4.3 Adding the Protocol

Now that you have created your protocol class, you must make the HoneyRJ application aware of your protocol. This process is simple and requires adding several lines of code to the application's main() method. Open the file src/honeyrj/HoneyRJMain.java. Inside the main() method, add the following lines after line 17, *createSampleProtocols()*.:

- LIProtocol yourProtocol = **new** [NAMEOFCLASS]();
  - Replace [NAMEOFCLASS] with the name of your class, for example, FtpProtocol.
  - For the FTP protocol, the line ends with new FTProtocol();
- LIModule yourModule = **new** LIModule(yourProtocol);
  - This line requires no modification. It creates the LIModule that will wrap your protocol class and provide socket communication.
- gui.AddModule(yourModule);
  - This line requires no modification. It adds your protocol to the HoneyRJ and the HoneyRJ's GUI.

At this point, you have successfully added a new protocol to HoneyRJ. You can run the HoneyRJMain.java file as a Java application and view the results.

In summary, the process for creating an additional protocol in HoneyRJ consists of making several design decisions and implementing the LIProtocol interface according to the answers to the design decisions. Once the class is created, several simple steps are followed to add the new protocol to the application.

---

5. Summary

Traditionally, IDS's have been used by network administrators to actively monitor network traffic for unauthorized activity. However, in today's world of increasingly encrypted connections, which intrusion detection systems are unable to monitor, honeypots have become an increasingly attractive alternative to locate sources of malicious traffic.

Honeypots, first created in 1998, function by recording all connections and connection attempts. A honeypot system should be placed on an unused IP address, such that no legitimate connection attempt would ever be directed to the honeypot. Two main types of honeypots are available today: high-interaction and low-interaction. Low-interaction honeypots are simple and provide partial implementations of common protocols, with the goal of recording only the source of malicious traffic. High-interaction honeypots are more complex and often are regular servers with advanced monitoring software and have the goal of helping researchers understand hacker's internal thought processes.

Honeypots are still an advancing field of computer science, with recent developments creating world-wide

networks of honeypots, commonly referred to as honeynets and distributed honeypots.

We provide an implementation case study of a low-interaction honeypot in Java, called HoneyRJ. HoneyRJ provides the basic logging and extendibility one would expect in a modern low-interaction honeypot.

Application, Source Code, Source Code Documentation and User Manual

- HoneyRJ Application (JAR)
- Source Code (zip)
- Source Code Documentation (JavaDoc HTML)
- Source Code Documentation (zip)
- User Manual (html)

References

[JavaTutorial2008] The Java Tutorials. "Writing the Server Side of a Socket" 2008. http://java.sun.com/docs/books/tutorial/networking/sockets/clientServer.html

This section of the Java tutorials provides an overview of how the socket API works in Java, in addition to sample code for both a single and multi-threaded server application.

[Anagnostakis05] Anagnostakis, K. G., et al. "Detecting targeted attacks using shadow honeypots." Proceedings of the 14th conference on USENIX Security Symposium. ACM, 2005. 129-144.

The authors, researchers at the University of Pennsylvania, Columbia University, and FORTH, present a hybrid system called a Shadow Honeypot. The shadow honeypot partially mirrors the state of a production system. The paper can be found at http://ics.forth.gr/dcs/Activities/papers/replay.pdf.

[Antonatos07] Antonatos, Spiros, Kostas Anagnostakis and Evangelos Markatos. "Honey@home: a new approach to large-scale threat monitoring." Proceedings of the 2007 ACM workshop on Recurring malcode. ACM, 2007. 38 - 45.

The paper describes the technical details of the Honey@home project located at http://www.honeyathome.org/. It addresses the challenges faced when designing and implementing a distributed honeypot architecture. The paper can be found at http://www.ics.forth.gr/dcs/Activities/papers/worm07_honeyathome.pdf.

[Cohen98] Cohen, Fred. "The Deception ToolKit." The Risks Digest 9 March 1998.

Cohen's initial public announcement of the Deception Toolkit, one of the first honeypots. It colorfully describes the author's views of the advantages honeypots provide. The announcement can be found at http://catless.ncl.ac.uk/Risks/19.62.html#subj11

[OpenDoor07] Open Door Networks, Inc. "Who's There? Firewall Advisor User's Guide" 2007. http://www.opendoor.com/WhosThere/UG/WTAppendix.html

[Euro07] European Network of Affined Honeypots. "Shadow Honeypots." 2007.

An extension of Anagnostakis' work on shadow honeypots. It shares many of the same sections and figures with the conference paper. The article can be found at http://www.fp6-noah.org/publications/.

[He04] He, Xing-YuLam, Kwok-Yan, et al. "Real-Time Emulation of Intrusion Victim in HoneyFarm." Content Computing. Springer Berlin / Heidelberg, 2004. 143-154.

Presents a novel method for allowing perceived outbound traffic in honeyfarms. Located at http://www.springerlink.com/content/dq3t9btumw6gvrk5/.

[hnet08] honeynet Project. "Know Your Enemy: Honeynets." 24 March 2008. http://old.honeynet.org/papers/honeynet/

An overview of honeypots with a good discussion of their advantages and disadvantages.

[Hussain03] Hussain, Alefiya, John Heidemann and Christos Papadopoulos. "A framework for classifying denial of service attacks." Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications. ACM, 2003. 99 - 110.

Provides a method for classifying and preventing denial of service attacks using an intrusion detection system. The paper is located at http://www.sigcomm.org/sigcomm2003/papers/p99-hussain.pdf

[Kemmerer02] Kemmerer, R.A. and G. Vigna. "Intrusion detection: a brief history and overview." Computer 2002: 27-30.

A very brief (4 page) introduction to intrusion detection systems in plain English. Discusses misuse and anomaly detection. Article found at http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1012428.

[Mukherjee94] Mukherjee, B., L. Heberlein and K. Levitt. "Network Intrusion Detection." IEEE Network May/Jun 1994: 26-41.

A survey of types of intrusion detection systems. Though outdated, the core discussion is still relevant.

[Paxson99] Paxson, Vern. "Bro: A System for Detecting Network Intruders in Real-Time." Computer Networks. 1999. 2435-2463.

Paxson's journal article describing the first system that could perform network intruder detect in real-time. The paper introduces a language for describing high-level network use rules. The article can be found at http://www.ece.cmu.edu/~adrian/731-sp04/readings/paxson99-bro.pdf.

[pcmag09] honeypot Definition - PC Magazine. pcmag.com. 24 March 2009. http://www.pcmag.com/encyclopedia_term/0,2542,t=honeypot&i=44335,00.asp

PC Magazine's encyclopedia entry for honeypot.

[Provos04] Provos, Niels. "A Virtual Honeypot Framework." In Proceedings of the 13th USENIX Security Symposium. 2004. 1-14.

The paper that laid the groundwork for the honeyd project. Provos describes building virtual honeypots which meet help honeypots meet the need to monitor a large network address space. The paper is located at http://www.citi.umich.edu/techreports/reports/citi-tr-03-1.pdf.

[Provos07] Provos, Niels and Thorsten Holz. Virtual Honeypots: From Botnet Tracking to Intrusion Detection. Addison-Wesley Professional, 2007.

A comprehensive book about honeypots, including ethical and legal issues. Has a perfect "5 star" rating on Amazon.com at the time of writing.

[Ptacek98] Ptacek, Thomas H. and Timothy N. Newsham. "Insertion, evasion, and denial of service: Eluding network intrusion detection." 1998.

An "old" survey of methods to circumvent or compromise intrusion detection systems. The article is available online at http://insecure.org/stf/secnet_ids/secnet_ids.html.

[Spitzner02] Spitzner, Lance. Honeypots: Tracking Hackers. Addison-Wesley Professional, 2002.

An older book providing a comprehensive discussion of honeypots. Includes an in-depth treatment of 6 available honeypots.

[Talabis07a] Talabis, Ryan. "Honeypots 101: A Brief History of Honeypots." 2007.

A fairly recent history of honeypots. The article does not provide in-depth information about events and implementations, but is a good starting point for a thorough survey of honeypot history. Available from here

[Talabis07b] Talabis, Ryan. "Honeypots 101: A Honeypot By Any Other Name." 2007.

A non-technical introduction to honeypots. Provides helpful analogies for understanding the way honeypots work. Available here.

[Verwoerd02] Verwoerd, Theuns and Ray Hunt. "Intrusion detection techniques and approaches." Computer Communications 15 September 2002: 1356-1365.

A description of machine learning approaches to network intrusion detection. The paper is available at http://sureserv.com/technic/datum_detail.php?id=468

[Wagner02] Wagner, David and Paolo Soto. "Mimicry Attacks on Host-Based Intrusion Detection Systems." Proceedings of the 9th ACM conference on Computer and communications security. ACM, 2002. 255 - 264.

A paper providing an example of an attack on a intrusion detection system. The paper provides an impetus for finding alternatives to traditions intrusion detection systems. Available at http://www.scs.carleton.ca/~soma/id-2007w/readings/wagner-mimicry.pdf

[Yang04] Yang, Geng, Rong Chunming and Yunping Dai. "A Distributed Honeypot System for Grid Security." Lecture Notes in Computer Science (2004): 1083-1086.

Yang et al. describe how to protect a grid of computers using a distributed honeypot framework.

[Yegneswaran07] Yegneswaran, V., et al. "Camouflaging Honeynets." In Proceedings of IEEE Global Internet Symposium. 2007.

A conference article that describes the construction of an intermediary framework for minimizing the chance of an attacker detecting a honeypot. Paper available at http://pages.cs.wisc.edu/~pb/gi07_final.pdf.

[JavaDocumentation2004] Java 2 Platform Standard Edition 5.0 Documentation. "Class ServerSocket" 2004. http://java.sun.com/j2se/1.5.0/docs/api/java/net/ServerSocket.html

This section of the Java documentation describes how the ServerSocket class, part of the Java sockets API.

[Eclipse08] Eclipse Project. "Import Wizard" 2008. http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.platform.doc.user/reference/ref-70.htm

This section of the Eclipse documentation describes how to import projects. It is referenced as part implementing a protocol.

[MSKB294676] Microsoft Knowledge Base Article #294676. http://support.microsoft.com/kb/294676

This Microsoft Knowledge Base article describes how to use the Run As command to give an application elevates privileges, such as those required by HoneyRJ.

[MsTechNet] Microsoft Technet - Netstat. http://technet.microsoft.com/en-us/library/bb490947.aspx

This section of the Microsoft TechNet documentation provides detail on usage of the netstat command, which can be used to verify HoneyRJ is working properly.

[Tail4Win]	Tail for Win32. http://tailforwin32.sourceforge.net/ *Tail4Win is a program that allows a user to monitor changes to text based files in real-time.*

Acronyms

IDS	Intrusion Detection System
DTK	Deception ToolKit
ADS	Anomaly Detection Systems
IDE	Integrated Development Environment
FTP	File Transfer Protocol
DOS	Denial of Service [attack]

Last Modified: April, 15 2008
This and other papers on latest advances in network security are available on line at http://www.cse.wustl.edu/~jain/cse571-09/index.html


Back to Raj Jain's Home Page