Honeyd: A Virtual Honeypot Daemon (Extended Abstract)

Niels Provos
Center for Information Technology Integration
University of Michigan
provos@citi.umich.edu

Abstract

Honeypots are closely monitored network decoys serving several purposes: they can distract adversaries from more valuable machines on a network, they can provide early warning about new attack and exploitation trends and they allow in-depth examination of adversaries during and after exploitation of a honeypot. Deploying physical honeypots is often time intensive and expensive as different operating systems require specialized hardware and every honeypot requires its own physical system. This paper presents Honeyd*, a framework for virtual honeypots, that simulates virtual computer systems at the network level. The simulated computer systems appear to run on unallocated network addresses. To fool network fingerprinting tools, Honeyd simulates the networking stack of different operating systems and can provide arbitrary services for an arbitrary number of virtual systems. Furthermore, the system supports virtual routing topologies that allow the creation of large virtual networks including characteristics like latency and packet loss. We discuss Honeyd's design and implementation.*

1 Introduction

Computer security is increasing in importance as more business is conducted over the Internet. Despite decades of research and experience, we are still unable to make secure computer systems or even measure their security.

As a result, exploitation of newly discovered vulnerabilities often catches us by surprise. Due to exploit automation and massive global scanning for vulnerabilities, adversaries are often able to compromise computer systems shortly after vulnerabilities become known [8].

One way of getting early warning of new vulnerabilities is to install computer systems on a network that we expect to be broken into. These systems have no other legitimate function and every attempt to connect to them is suspect. We call such a system a *honeypot*. It may run any operating system and any number of services. The configured services determine the venues an adversary may choose to compromise the system. A *high-interaction* honeypot simulates all aspects of an operating system, whereas a *low-interaction* honeypots simulates only some parts, for example the network stack [7]. We also differentiate between *physical* and *virtual* honeypots. A physical honeypot exists as a machine with a corresponding IP address on the network whereas a virtual honeypot is hosted on another machine that responds to network traffic directed to the virtual honeypot.

Virtual honeypots are attractive because they do not require additional computer systems. Using virtual honeypots, it is possible to populate a network with hosts running a variety of different operating systems. However, to convince adversaries that a virtual honeypot is running a certain operating system, it is necessary to simulate the TCP/IP stack of the target operating system carefully. In other words, we need to be able to fool TCP/IP stack fingerprinting tools like *Xprobe* [1] or *Nmap* [4].

This paper provides a brief overview of the design and implementation of *Honeyd*, a daemon that simulates the TCP/IP stack of operating systems to create virtual honeypots. Honeyd supports TCP, UDP and ICMP. It listens to network requests destined for its configured virtual honeypots. Honeyd responds according to the services that run on the virtual honeypot. Before sending a response packet to the network, the packet is modified by Honeyd's personality engine to match the network behavior of the configured operating system personality.

To simulate more realistic networks, Honeyd supports the creation of virtual network topologies. The networks can be configured to contain routers with configurable link characteristics like latency and packet loss. When using tools like *traceroute*, the network traffic appears to follow the configured topology.

We present an experimental evaluation of Honeyd that shows how fingerprinting tools like Nmap detect the configured operating system and services. Furthermore, we evaluate the support of virtual network topologies with tools like traceroute.

The rest of this paper is organized as follows. Section 2 discusses the design and implementation of Honeyd. In Section 3, we evaluate the implementation and show that Honeyd fools fingerprinting tools in practice. We present related work in Section 4. We conclude in Section 5.

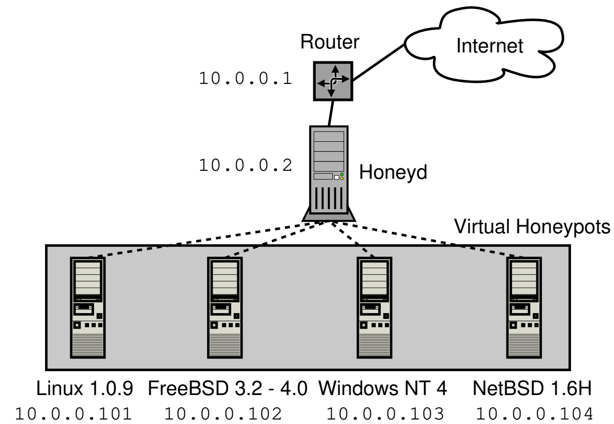

Figure 1: Honeyd receives traffic for its virtual honeypots via a router or Proxy ARP. For each honeypot, Honeyd can simulate the network stack behavior of a different operating system.

2 Design and Implementation

In this section, we discuss the design and implementation of Honeyd. We discuss our intended goals and show how they were implemented.

We expect adversaries to interact with our honeypots only at the network level. Instead of simulating every aspect of an operating system, we decided to simulate only its network stack. The main drawback of this approach is that an adversary never gains access to a complete system even if he compromises a simulated service. On the other hand, we are still able to capture connection and compromise attempts. For that reason, Honeyd is a low-interaction virtual honeypot that simulates TCP and UDP services.

Honeyd must be able to handle virtual honeypots on multiple IP addresses simultaneously. This allows us to populate the network with a number of virtual honeypots that can simulate different operating systems and services. Furthermore, Honeyd must be able to simulate different network topologies.

Honeyd is implemented as a Unix daemon that runs on a workstation and listens to network traffic; see Figure 1. Before we give an overview of the Honeyd architecture, we explain how network packets for virtual honeypots reach the Honeyd host.

2.1 Receiving Network Data

Honeyd is designed to reply to network packets that have the destination IP address of one of the honeypots that it simulates. We need to configure the network appropriately for Honeyd to actually receive such packets. To do this, we either create a special route at the router for the virtual IP addresses or use Proxy ARP [3].

In the following, we assume that A is the IP address of our router and that B is the IP address of the Honeyd host. The IP addresses of virtual honeypots need to fall within our local network. We designate them $V_1, \ldots, V_n$. When an adversary sends a packet from the Internet to honeypot V_i, router A receives it first and attempts to forward it. The router queries its routing table to find the forwarding address for V_i. There are three possible results: the router drops the packet because there is no route to V_i, router A forwards the packet to another router, or V_i falls in local network range of the router and thus is directly reachable by A.

We make use of the latter two cases to direct traffic for V_i to B. The easiest way is to configure routing entries for V_i with $1 \leq i \leq n$ that point to B. In that case, the router forwards packets for our virtual honeypots directly to the Honeyd host. If no special route has been configured, the router uses ARP to determine the MAC address of the virtual honeypot. As there is no corresponding physical machine, the ARP requests remain unanswered and the router drops the packet after a few retries. We configure the Honeyd host to reply to ARP requests for V_i with its own MAC addresses. This is called Proxy ARP and allows the router to send packets for V_i to B's MAC address.

2.2 Honeyd Architecture

In this section, we give an overview of Honeyd's architecture; see Figure 2.

When the Honeyd daemon receives a packet for one of the virtual honeypots, it is processed by a central packet dispatcher. The dispatcher checks the length of the IP packet and verifies its checksum. The daemon knows only three protocols: ICMP, TCP and UDP [9]. Packets for other protocols are discarded.

The dispatcher queries the configuration database for a honeypot configuration that corresponds to the destination IP address. If no such configuration exists, the default template is used. Then the dispatcher calls the protocol specific handler with the received packet and the corresponding honeypot configuration. For ICMP, the only packet that is currently supported is the `ICMP_ECHO` request. The daemon answers with an `ICMP_ECHO` reply packet.

For TCP and UDP, the daemon can establish connections to arbitrary services. Services are external programs that receive data on *stdin* and send their output to *stdout*. When a connection request is received, the daemon checks if the packet is part of an established connection. In that case, any new data is sent to the already started service program. If the packet contains a connection request, a new process is created to run the appropriate service.

Honeyd contains a simplified TCP state machine, *i.e* the three-way handshake for connection establishment and connection teardown via `FIN` or `RST` are fully supported. However, receiver and congestion window management is not fully implemented.

An UDP packet to a closed port is correctly answered with an ICMP *port unreachable* message. This allows tools like traceroute to work correctly.

Instead of establishing a connection with a service program, the daemon also supports dynamic redirection of the service. This allows us to forward a connection request for a web server running on a virtual honeypot to a real web server. It is also possible to redirect connections to the adversary herself, *e.g.* a redirected SSH connection might cause an adversary to attempt to compromise her own SSH server.

Before any packet is sent to the network, it is processed by the personality engine. It adjusts the packet's content so that it seems to originate from the network stack of the configured operating system; see Section 2.4 for more details.

2.3 Routing Topology

Instead of simulating a flat network, Honeyd also supports virtual routing topologies. We can no longer use Proxy ARP for the packets to reach the Honeyd host, but need to configure a router to delegate a network range to our host. This network range can be split into sub-networks. Currently, the virtual routing topology is restricted to a rooted tree. The root of the tree is the point at which packets enter the virtual routing topology.

Each non-terminal node of the tree represents a router and each edge a link that contains latency and packet loss as attributes. Each terminal node of the

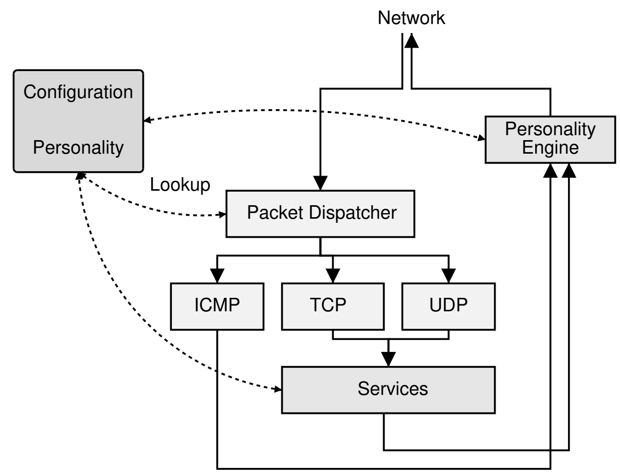

Figure 2: This diagram gives an overview of Honeyd's architecture. Incoming packets are dispatched to the correct protocol handler. For TCP and UDP, the configured services receive new data and send responses if necessary. All outgoing packets are modified by the personality engine to mimic the behavior of the configured network stack.

tree corresponds to a network.

When the daemon receives a packet, it traverses the tree starting at the root until it finds a node that contains the destination IP address of the packet. The packet loss and latency of all edges on the path is accumulated and determines if the packet is dropped and for how long its delivery should be delayed.

The daemon also decrements the *time to live* (TTL) of the packet for each traversed router. If the TTL reaches zero, the daemon sends an ICMP *time exceeded* message with the source IP address of the router that causes the TTL to reach zero.

2.4 Personality Engine

Honeyd uses the term *personality* to refer to the network stack behavior of a virtual honeypot. The daemon uses the Nmap fingerprint list as a reference. Each fingerprint has a format similar to the following example:

```
Fingerprint IRIX 6.5.15m on SGI O2
TSeq(Class=TD%gcd=<104%SI=<1AE%IPID=I%TS=2HZ)
T1(DF=N%W=EF2A%ACK=S++%Flags=AS%Ops=MNWNNTNNM)
T2(Resp=Y%DF=N%W=0%ACK=S%Flags=AR%Ops=)
T3(Resp=Y%DF=N%W=EF2A%ACK=O%Flags=A%Ops=NNT)
T4(DF=N%W=0%ACK=O%Flags=R%Ops=)
T5(DF=N%W=0%ACK=S++%Flags=AR%Ops=)
T6(DF=N%W=0%ACK=O%Flags=R%Ops=)
T7(DF=N%W=0%ACK=S%Flags=AR%Ops=)
PU(Resp=N)
```

We use the string after the *Fingerprint* token as the personality name. The lines after the name describe test results for nine different tests. The first test is the most comprehensive. It determines how the network stack of the remote operating system creates the *initial sequence number* (ISN) for TCP SYN segments. Nmap indicates the difficulty of predicting ISNs in the *Class* field. Predictable ISNs are a long known security problem because they allow an adversary to spoof connections [2]. The *gcd* and *SI* field provide more detailed information about the ISN distribution. The first test also determines how IP identification numbers and TCP timestamps are generated.

The next seven tests determine the stack's behavior for packets that arrive on open and closed TCP ports. The last test analyzes the ICMP response packet to a closed UDP port.

Honeyd keeps state for each honeypot. This includes information about ISN generation, the boot time of the honeypot and the current IP identification number.

Before a packet is sent to the network, it passes through the personality engine.

For ICMP packets, the protocol part of the packet is currently changed only if it is of type *destination unreachable* and code *port unreachable*; see Figure 3. The daemon looks up the *PU* test entry for the personality to determine how the quoted IP header needs to be modified. Many operating systems modify the incoming packet by changing fields from network to host order and as a result quote the IP and UDP header incorrectly. Honeyd introduces these errors if necessary.

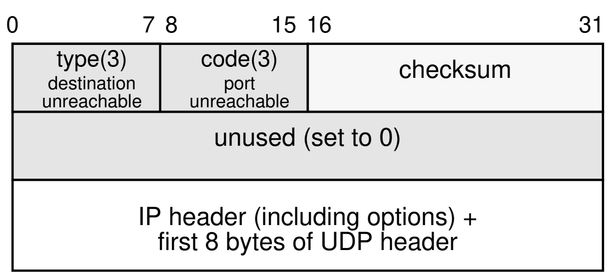

Figure 3: The diagram shows the structure of an ICMP port unreachable message. Honeyd introduces errors into the quoted IP header to match the behavior of network stacks.

Nmap's fingerprinting is mostly concerned with the operating system's TCP implementation. TCP is a stateful connection-oriented protocol that provides error recovery and congestion control [5]. It also supports additional options that are not implemented by all systems. The size of the advertised receiver windows varies between implementations and is used by Nmap as part of the fingerprint.

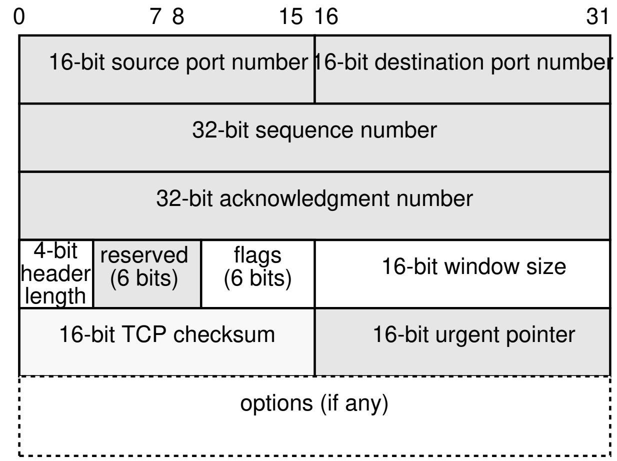

Figure 4: The diagram shows the structure of the TCP header. Honeyd changes options and other parameters to match the behavior of network stacks.

When the daemon sends a packet for a not yet established TCP connection, it takes the initial window size from the Nmap fingerprint. After a connection has been established, the daemon adjusts the window size according to the amount of buffered data.

If the fingerprint includes TCP options, Honeyd inserts them into a packet as long as they have been correctly negotiated during connection establishment. For TCP timestamps, the daemon uses the fingerprint to determine the frequency with which the timestamp is updated. For most operating systems, the update frequency is 2 Hz.

Generating a matching distribution of initial sequence numbers is more difficult. Nmap obtains six ISN samples and analyzes their consecutive differences. Nmap recognizes several ISN generation types: constant differences, differences that are multiples of a constant, completely random differences, time dependent and random increments. To differentiate between the latter two cases, Nmap calculates the greatest common divisor (gcd) and standard deviation for the collected differences.

For each honeypot, the daemon keeps track of the last ISN that was generated and its generation time. For new TCP connection requests, Honeyd uses a formula that approximates the distribution described by the fingerprint's gcd and standard deviation. As a result, the generated ISNs match the generation class that Nmap expects for the particular operating system.

For the IP header, Honeyd adjusts the generation of the identification number. It can either be zero, increment by one, or random.

```
route entry 10.0.0.1
route 10.0.0.1 link 10.0.0.0/24
route 10.0.0.1 add net 10.1.0.0/16 10.1.0.1 latency 55ms loss 0.1
route 10.0.0.1 add net 10.2.0.0/16 10.2.0.1 latency 20ms loss 0.1
route 10.1.0.1 link 10.1.0.0/24
route 10.2.0.1 link 10.2.0.0/24

create routerone
set routerone personality "Cisco 7206 running IOS 11.1(24)"
set routerone default tcp action reset
add routerone tcp port 23 "scripts/router-telnet.pl"

create netbsd
set netbsd personality "NetBSD 1.5.2 running on a Commodore Amiga (68040 processor)"
set netbsd default tcp action reset
add netbsd tcp port 22 proxy $ipsrc:22
add netbsd tcp port 80 "sh scripts/web.sh"

bind 10.0.0.1 routerone
bind 10.1.0.2 netbsd
```

Figure 5: An example configuration for Honeyd. The configuration language is a context-free grammar. This example creates a virtual routing topology and defines two templates: a router that can be accessed via telnet and a host that is running a web server.

2.5 Configuration

Virtual honeypots are configured via templates. A template is a reference for a completely configured computer system. New templates are created with the *create* command.

The *set* and *add* commands change the configuration of a template. Using the set command, we assign a personality from the Nmap fingerprint file to a template. The personality determines the behavior of the network stack as discussed in Section 2.4. The set command is also used to define the default behavior for the supported network protocols. The default behavior can be one of the following values: *block*, *reset*, or *open*. Block means that all packets for the specified protocol are dropped by default, reset indicates that all ports are closed by default and open means that they are all open by default. The latter two settings make a difference only for UDP and TCP.

Using the add command, we specify the services that are remotely accessible. Besides the template name, we need to specify the protocol, port and the command to execute for each service. Instead of specifying a service, Honeyd also recognizes the keyword *proxy* that allows us to forward network connections to a different host. The daemon expands the following four variables for both the service and the proxy statement: *$ipsrc*, *$ipdst*, *$sport*, and *$dport*. This allows services to adapt their behavior depending on the particular network connection they are handling. It is also possible to redirect network probes back to the host that is probing us.

The *bind* command is used to assign a template to an IP address. If no template has been assigned to an IP address, the *default* template is used. Figure 5 shows an example configuration that specifies a routing topology and two templates. The router template mimics the network stack of a Cisco 7206 router and is accessible only via telnet. Whereas the web server template runs two services: a simple web server and a forwarder for SSH connections. In this case, the forwarder redirects SSH connections back to the connection initiator.

3 Evaluation

This section presents a brief evaluation of Honeyd's ability to create virtual network topologies and to mimic different network stacks.

We start Honeyd with a similar configuration to the one shown in Figure 5 and use traceroute to find the routing path to a virtual host. We notice that the measured latency is double the latency that we configured which is the correct time because packets have to travel each link twice.

Running Nmap against the two IP addresses `10.0.0.1` and `10.1.0.2` results in the correct identification of the configured personalities. Nmap states

```
$ traceroute -n 10.3.0.10
traceroute to 10.3.0.10 (10.3.0.10), 64 hops max
 1  10.0.0.1  0.456 ms  0.193 ms  0.93 ms
 2  10.2.0.1  46.799 ms  45.541 ms  51.401 ms
 3  10.3.0.1  68.293 ms  69.848 ms  69.878 ms
 4  10.3.0.10  79.876 ms  79.798 ms  79.926 ms
```

Figure 6: Using traceroute, we measure a routing path in the virtual routing topology. The measured latencies match the configured ones.

that `10.0.0.1` seems to be a Cisco router and that `10.1.0.2` seems to run NetBSD. Xprobe identifies `10.0.0.1` as Cisco router and lists a numer of possible operating systems including NetBSD for `10.1.0.2`. A more thorough evaluation of Honeyd is the subject of future work.

4 Related Work

There are several areas of research in TCP/IP stack fingerprinting, among them: effective methods to determine the remote operating system either by active probing or by passive analysis of network traffic, and defeating TCP/IP stack fingerprinting by normalizing network traffic.

Fyodor's Nmap uses TCP and UDP probes to determine the operating system of a host [4]. Nmap collects the responses of a network stack to different queries and matches them to a signature database to determine the operating systems of the queried host. Nmap's fingerprint database is extensive and we use it as the reference for operating system personalities in Honeyd.

Instead of actively probing a remote host to determine its operating systems, it is possible to identify the remote operating system by passively analyzing its network packets as done by the passive OS fingerprinting tool *P0f* [10]. The TCP/IP flags inspected by P0f are similar to the data collected in Nmap's fingerprint database.

On the other hand, Smart *et al.* show how to defeat fingerprinting tools by scrubbing network packets so that artifacts identifying the remote operating system are removed [6]. This approach is similar to Honeyd's personality engine as both systems change network packets to influence fingerprinting tools. In contrast to the fingerprint scrubber that removes identifiable information, Honeyd changes network packets in such a way that they contain artifacts of the configured operating system.

5 Conclusion

We presented Honeyd, a framework for creating virtual honeypots. Honeyd mimics the network stack behavior of operating systems to fool fingerprinting tools like Nmap.

We gave a brief overview of Honeyd's design and implementation. Our evaluation shows that Honeyd is effective in creating virtual routing topologies and successfully fools fingerprinting tools.

Honeyd is freely available as source code and can be downloaded from `http://www.citi.umich.edu/u/provos/honeyd/`.

6 Acknowledgments

I would like to thank Dug Song, Jamie Van Randwyk and Eric Thomas for helpful suggestions and contributions. I also thank Therese Pasquesi and Jose Nazario for careful reviews.

References

[1] Ofir Arkin and Fyodor Yarochkin. Xprobe v2.0: A "Fuzzy" Approach to Remote Active Operating System Fingerprinting. `http://www.xprobe2.org/`, August 2002.

[2] Steven M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communications Review*, 19:2:32–48, 1989.

[3] Smoot Carl-Mitchell and John S. Quarterman. Using ARP to Implement Transparent Subnet Gateways. RFC 1027, October 1987.

[4] Fyodor. Remote OS Detection via TCP/IP Stack Fingerprinting. `http://www.nmap.org/nmap/nmap-fingerprinting-article.html`, October 1998.

[5] Jon Postel. Transmission Control Protocol. RFC 793, September 1981.

[6] Matthew Smart, G. Robert Malan, and Farnam Jahanian. Defeating TCP/IP Stack Fingerprinting. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[7] Lance Spitzner. *Honeypots: Tracking Hackers*. Addison Wesley Professional, September 2002.

[8] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to Own the Internet in your Spare Time. In *Proceedings of the 11th USENIX Secuirty Symposium*, August 2002.

[9] W. R. Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley, 1994.

[10] Michal Zalewski and William Stearns. Passive OS Fingerprinting Tool. `http://www.stearns.org/p0f/README`. Viewed on 12th January 2003.