CITI Technical Report 03-1

A Virtual Honeypot Framework

Niels Provos
`niels@google.com`

Abstract

A honeypot is a closely monitored network decoy serving several purposes: it can distract adversaries from more valuable machines on a network, can provide early warning about new attack and exploitation trends, or allow in-depth examination of adversaries during and after exploitation of a honeypot. Deploying a physical honeypot 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 deceive network fingerprinting tools, Honeyd simulates the networking stack of different operating systems and can provide arbitrary routing topologies and services for an arbitrary number of virtual systems. This paper discusses Honeyd's design and shows how the Honeyd framework helps in many areas of system security, *e.g.* detecting and disabling worms, distracting adversaries, or preventing the spread of spam email.

October 21, 2003

Center for Information Technology Integration
University of Michigan
535 West William Street
Ann Arbor, MI 48103-4943

A Virtual Honeypot Framework

Niels Provos*
Google, Inc.
niels@google.com

1 Introduction

Internet security is increasing in importance as more and more business is conducted there. Yet, 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. Exploit automation and massive global scanning for vulnerabilities enable adversaries to compromise computer systems shortly after vulnerabilities become known [23].

One way to get early warnings of new vulnerabilities is to install and monitor computer systems on a network that we expect to be broken into. Every attempt to contact these systems via the network is suspect. We call such a system a *honeypot*. If a honeypot is compromised, we study the vulnerability that was used to compromise it. A honeypot may run any operating system and any number of services. The configured services determine the vectors an adversary may choose to compromise the system.

A physical honeypot is a real machine with its own IP address. A virtual honeypot is simulated by another machine that responds to network traffic sent to the virtual honeypot.

Virtual honeypots are attractive because they requirer fewer computer systems, which reduces maintenance costs. Using virtual honeypots, it is possible to populate a network with hosts running numerous operating systems. To convince adversaries that a virtual honeypot is running a given operating system, we need to simulate the TCP/IP stack of the target operating system carefully, in order to fool TCP/IP stack fingerprinting tools like *Xprobe* [1] or *Nmap* [7].

This paper describes the design and implementation of *Honeyd*, a framework for virtual honeypots that simulates computer systems at the network level. Honeyd supports the IP protocol suites [24] and responds to network requests for its virtual honeypots according to the services that are configured for each virtual honeypot. When sending a response packet, Honeyd's personality engine makes it match the network behavior of the configured operating system personality.

To simulate real networks, Honeyd creates virtual networks that consist of arbitrary routing topologies with configurable link characteristics such as latency and packet loss. When networking mapping tools like *traceroute* are used to probe the virtual network, they discover only the topologies simulated by Honeyd.

Our experimental evaluation of Honeyd verifies that fingerprinting tools are deceived by the simulated systems and that our virtual network topologies seem realistic to network mapping tools.

To demonstrate the power of the Honeyd framework, we show how it can be used in many areas of system security. For example, Honeyd can help with detecting and disabling worms, distracting adversaries, or preventing the spread of spam email.

The rest of this paper is organized as follows. Section 2 presents background information on honeypots. In Section 3, we discuss the design and implementation of Honeyd. Section 4 presents an evaluation of the Honeyd framework in which we verify that fingerprinting and network mapping tools are fooled to report the specified system configurations. We describe how Honeyd can help to improve system security in Section 5 and present related work in Section 6. We summarize and conclude in Section 7.

2 Honeypots

This section presents background information on honeypots and our terminology. We provide motivation for their use by comparing honeypots to network intrusion detection systems (NIDS) [17]. The amount of useful information provided by NIDS is decreasing in the face of ever more sophisticated evasion techniques [19, 26] and an increasing number of protocols that employ encryption to protect network traffic from eavesdroppers. NIDS also suffer from high false positive rates that decrease their usefulness even further.

*This research was conducted by the author while at the Center for Information Technology Integration of the University of Michigan.

Honeypots can help with some of these problems.

A *honeypot* is as a closely monitored computing resource that we intend to be probed, attacked, or compromised. The value of a honeypot is determined by the information that we can obtain from it. Monitoring the data that enters and leaves a honeypot lets us gather information that is not available to NIDS. For example, we can log the key strokes of an interactive session even if encryption is used to protect the network traffic. To detect malicious behavior, NIDS require signatures of known attacks and often fail to detect compromises that were unknown at the time it was deployed. On the other hand, honeypots can detect vulnerabilities that are not yet understood. For example, we can detect compromise by observing network traffic leaving the honeypot even if the means of the exploit has never been seen before.

Because a honeypot has no production value, any attempt to contact it is suspicious. Consequently, forensic analysis of data collected from honeypots is less likely to lead to false positives than data collected by NIDS.

Honeypots can run any operating system and any number of services. The configured services determine the vectors available to an adversary for compromising or probing the system. A *high-interaction* honeypot simulates all aspects of an operating system. A *low-interaction* honeypots simulates only some parts, for example the network stack [22]. A high-interaction honeypot can be compromised completely, allowing an adversary to gain full access to the system and use it to launch further network attacks. In contrast, low-interaction honeypots simulate only services that cannot be exploited to get complete access to the honeypot. Low-interaction honeypots are more limited, but they are useful to gather information at a higher level, *e.g.*, learn about network probes or worm activity. They can also be used to analyze spammers or for active countermeasures against worms; see Section 5.

We also differentiate between *physical* and *virtual* honeypots. A physical honeypot is a real machine on the network with its own IP address. A virtual honeypot is simulated by another machine that responds to network traffic sent to the virtual honeypot.

When gathering information about network attacks or probes, the number of deployed honeypots influences the amount and accuracy of the collected data. A good example is measuring the activity of HTTP based worms [21]. We can identify these worms only after they complete a TCP handshake and send their payload. However, most of their connection requests will go unanswered because they contact randomly chosen IP addresses. A honeypot can capture the worm payload by configuring it to function as a web server. The more honeypots we deploy the more likely one of them is contacted by a worm.

Physical honeypots are often high-interaction, so allowing the system to be compromised completely, they are expensive to install and maintain. For large address spaces, it is impractical or impossible to deploy a physical honeypot for each IP address. In that case, we need to deploy virtual honeypots.

3 Design and Implementation

In this section, we present Honeyd, a lightweight framework for creating virtual honeypots. The framework allows us to instrument thousands of IP addresses with virtual machines and corresponding network services. We start by discussing design considerations, then describe Honeyd's architecture and implementation.

We limit adversaries to interacting with our honeypots only at the network level. Instead of simulating every aspect of an operating system, we choose 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. However, we can mitigate these drawbacks by combining Honeyd with a virtual machine like Vmware [25]. This is discussed in the related work section. For that reason, Honeyd is a low-interaction virtual honeypot that simulates TCP and UDP services. It also understands and responds correctly to ICMP messages.

Honeyd must be able to handle virtual honeypots on multiple IP addresses simultaneously, in order to populate the network with numerous virtual honeypots simulating different operating systems and services. To increase the realism of our simulation, the framework must be able to simulate arbitrary network topologies. To simulate address spaces that are topologically dispersed and for load sharing, the framework also needs to support network tunneling.

Figure 1 shows a conceptual overview of the framework's operation. A central machine intercepts network traffic sent to the IP addresses of configured honeypots and simulates their responses. Before we describe Honeyd's architecture, we explain how network packets for virtual honeypots reach the Honeyd host.

3.1 Receiving Network Data

Honeyd is designed to reply to network packets whose destination IP address belongs to one of the sim-

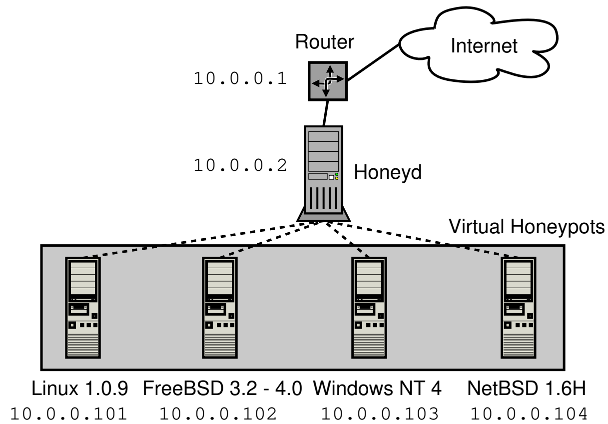

Figure 1: Honeyd receives traffic for its virtual honeypots via a router or Proxy ARP. For each honeypot, Honeyd can simulate the network stack behavior of a different operating system.

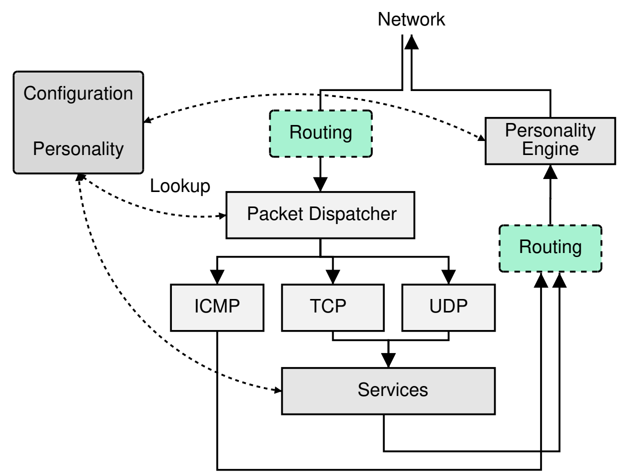

Figure 2: This diagram gives an overview of Honeyd's architecture. Incoming packets are dispatched to the correct protocol handler. For TCP and UDP, the configured services receive new data and send responses if necessary. All outgoing packets are modified by the personality engine to mimic the behavior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

ulated honeypots. For Honeyd, to receive the correct packets, the network needs to be configured appropriately. There are several ways to do this, *e.g.,* we can create special routes for the virtual IP addresses that point to the Honeyd host, or we can use Proxy ARP [3], or we can use network tunnels.

Let A be the IP address of our router and B the IP address of the Honeyd host. In the simplest case, the IP addresses of virtual honeypots lie within our local network. We denote them $V_1, \ldots, V_n$. When an adversary sends a packet from the Internet to honeypot V_i, router A receives and attempts to forward the packet. The router queries its routing table to find the forwarding address for V_i. There are three possible outcomes: the router drops the packet because there is no route to V_i, router A forwards the packet to another router, or V_i lies in local network range of the router and thus is directly reachable by A.

We use 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 ARPs to determine the MAC address of the virtual honeypot. As there is no corresponding physical machine, the ARP requests go 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.

In more complex environments, it is possible to tunnel network address space to a Honeyd host. We use the generic routing encapsulation (GRE) [9, 10] tunneling protocol described in detail in Section 3.4.

3.2 Architecture

Honeyd's architecture consists of several components: a configuration database, a central packet dispatcher, protocol handlers, a personality engine, and an optional routing component; see Figure 2.

Incoming packets are processed by the central packet dispatcher. It first checks the length of an IP packet and verifies the packet's checksum. The framework is aware of the three major Internet protocols: ICMP, TCP and UDP. Packets for other protocols are logged and silently discarded.

Before it can process a packet, the dispatcher must query the configuration database to find a honeypot configuration that corresponds to the destination IP address. If no specific configuration exists, a default template is used. Given a configuration, the packet and corresponding configuration is handed to the protocol specific handler.

The ICMP protocol handler supports most ICMP requests. By default, all honeypot configurations respond to *echo* requests and process *destination unreachable* messages. The handling of other requests depends on the configured personalities as described in Section 3.3.

For TCP and UDP, the framework can establish connections to arbitrary services. Services are external applications that receive data on *stdin* and send their output to *stdout*. The behavior of a service depends entirely on the external application. When a connection request is received, the framework checks if the packet is part of an established connection. In that case, any new data is sent to the already started service application. If the packet contains a connection request, a new process is created to run the appropriate service. Instead of creating a new process for each connection, the framework supports *subsystems*. A subsystem is an application that runs in the name space of the virtual honeypot. The subsystem specific application is started when the corresponding virtual honeypot is instantiated. A subsystem can bind to ports, accept connections, and initiate network traffic.

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

UDP datagrams are passed directly to the application. When the framework receives a UDP packet for a closed port, it sends an ICMP *port unreachable* message unless this is forbidden by the configured personality. In sending ICMP port unreachable messages, the framework allows network mapping tools like traceroute to discover the simulated network topology.

In addition to establishing a connection to a local service, the framework also supports redirection of connections. The redirection may be static or it can depend on the connection quadruple (source address, source port, destination address and destination port). Redirection lets us to forward a connection request for a service on a virtual honeypot to a service running on a real server. For example, we can redirect DNS requests to a proper name server. Or we can reflect connections back to an adversary, *e.g.* just for run we might redirect an SSH connection back to the originating host and cause the adversary to attack her own SSH server. *Evil laugh.*

Before a packet is sent to the network, it is processed by the personality engine. The personality engine adjusts the packet's content so that it appears to originate from the network stack of the configured operating system.

3.3 Personality Engine

Adversaries commonly run fingerprinting tools like *Xprobe* [1] or *Nmap* [7] to gather information about a target system. It is important that honeypots do not stand out when fingerprinted. To make them appear real to a probe, Honeyd simulates the network stack behavior of a given operating system. We call this the *personality* of a virtual honeypot. Different personalities can be assigned to different virtual honeypots. The personality engine makes a honeypot's network stack behave as specified by the personality by introducing changes into the protocol headers of every outgoing packet so that they match the characteristics of the configured operating system.

```
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)
```

Figure 3: An example of an Nmap fingerprint that specifies the network stack behavior of a system running IRIX.

The framework uses Nmap's fingerprint database as its reference for a personality's TCP and UCP behavior; Xprobe's fingerprint database is used as reference for a personality's ICMP behavior.

Next, we explain how we use the information provided by Nmap's fingerprints to change the characteristics of a honeypot's network stack.

Each Nmap fingerprint has a format similar to the example shown in Figure 3. We use the string after the *Fingerprint* token as the personality name. The lines after the name describe the 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 post a 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.

The framework keeps state for each honeypot. The state includes information about ISN generation, the boot time of the honeypot and the current IP packet identification number. Keeping state is necessary so that we can generate subsequent ISNs that follow the

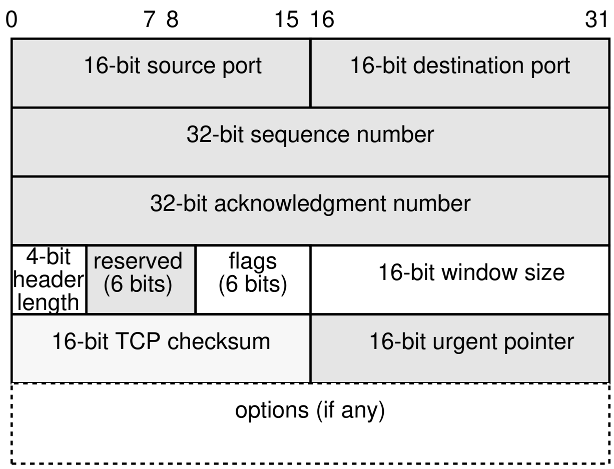

Figure 4: The diagram shows the structure of the TCP header. Honeyd changes options and other parameters to match the behavior of network stacks.

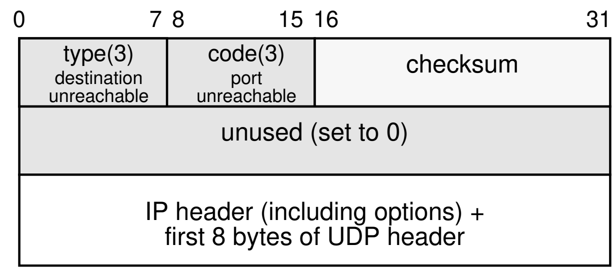

Figure 5: The diagram shows the structure of an ICMP port unreachable message. Honeyd introduces errors into the quoted IP header to match the behavior of network stacks.

distribution specified by the fingerprint.

Nmap's fingerprinting is mostly concerned with an operating system's TCP implementation. TCP is a stateful, connection-oriented protocol that provides error recovery and congestion control [18]. TCP also supports additional options, not all of which implemented by all systems. The size of the advertised receiver windows varies between implementations and is used by Nmap as part of the fingerprint.

When the framework sends a packet for a newly established TCP connection, it uses the Nmap fingerprint to see the initial window size. After a connection has been established, the framework adjusts the window size according to the amount of buffered data.

If TCP options present in the fingerprint have been negotiated during connection establishment, then Honeyd inserts them into the response packet. The framework uses the fingerprint to determine the frequency with which TCP timestamps are updated. For most operating systems, the update frequency is 2 Hz.

Generating the correct distribution of initial sequence numbers is tricky. 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.

The framework keeps track of the last ISN that was generated by each honeypot 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. In this way, 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.

For ICMP packets, the personality engine uses the *PU* test entry to determine how the quoted IP header should be modified using the associated Xprobe fingerprint for further information. Some 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 5 shows an example for an ICMP destination unreachable message. The framework also supports the generation of other ICMP messages, not described here due to space considerations.

3.4 Routing Topology

Honeyd can simulate arbitrary virtual routing topologies, so it is not always possible to use Proxy ARP to direct the packets to the Honeyd host. Instead, we need to configure routers to delegate network address space to our host.

Normally, the virtual routing topology is a tree rooted where packets enter the virtual routing topology. Each interior node of the tree represents a router and each edge a link that contains latency and packet loss characteristics. Terminal nodes of the tree correspond to networks. The framework supports multiple entry points that can exit in parallel. An entry router is chosen by the network space for which it is responsible.

To simulate an asymmetric network topology, we consult the routing tables when a packet enters the framework and again when it leaves the framework; see Figure 2. In this case, the network topology resembles a directed acyclic graph[1].

When the framework receives a packet, it finds the correct entry routing tree and traverses it, starting at the root until it finds a node that contains the destination IP address of the packet. Packet loss and latency of all edges on the path are accumulated to determine if the packet is dropped and how long its delivery should be delayed.

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

For network simulations, it is possible to integrate real systems into the virtual routing topology. When the framework receives a packet for a real system, it traverses the topology until it finds a virtual router that is directly responsible for the network space that the real machine belongs to. The framework sends an ARP request if necessary to discover the hardware address of the system, then encapsulates the packet in an Ethernet frame. Similarly, the framework responds with ARP replies from the corresponding virtual router when the real system sends ARP requests.

We can split the routing topology using GRE to tunnel networks. This allows us to load balance across several Honeyd installations by delegating parts of the address space to different Honeyd hosts. Using GRE tunnels, it is also possible to delegate networks that belong to separate parts of the address space to a single Honeyd host. For the reverse route, an outgoing tunnel is selected based both on the source and the destination IP address. An example of such a configuration is described in Section 5.

3.5 Configuration

A virtual honeypots is configured with a template, 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. The set command assigns a personality from the Nmap fingerprint file to a template. The personality determines the behavior of the network stack, as discussed in Section 3.3. The set command also defines the default behavior for the supported network protocols. The default behavior is 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. Open means that they are all open by default. The latter settings make a difference only for UDP and TCP.

We specify the services that are remotely accessible with the add command. In addition to 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 framework expands the following four variables for both the service and the proxy statement: *$ipsrc*, *$ipdst*, *$sport*, and *$dport*. Variable expansion allows a service to adapt its behavior depending on the particular network connection it is handling. It is also possible to redirect network probes back to the host that is doing the probing.

The *bind* command assigns a template to an IP address. If no template is assigned to an IP address, we use the *default* template. Figure 6 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. 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. A real machine is integrated into the virtual routing topology at IP address 10.1.0.3.

3.6 Logging

The Honeyd framework supports several ways of logging network activity. It can create connection logs that report attempted and completed connections for all protocols. More usefully, information can be gathered from the services themselves. Service applications can report data to be logged to Honeyd via *stderr*. The framework uses *syslog* to store the information on the system. In most situations, we expect that Honeyd runs in conjunction with a NIDS.

4 Evaluation

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

We start Honeyd with a configuration similar to the one shown in Figure 6 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. This is correct because packets have to traverse each link twice.

Running Nmap against IP addresses `10.0.0.1` and `10.1.0.2` results in the correct identification of the

[1]Although it is possible to configure routing loops, this is normally undesirable and should be avoided.

```
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
bind 10.1.0.3 to fxp0
```

Figure 6: 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. A real system is integrated into the virtual routing topology at IP address 10.1.0.3.

configured personalities. Nmap reports 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 number of possible operating systems, including NetBSD, for `10.1.0.2`.

To fully test if the framework fools Nmap, we set up a B-class network populated with virtual honeypots for every fingerprint in Nmap's fingerprint file. After removing duplicates, we found 600 distinct fingerprints. The honeypots were configured so that all but one port was closed; the open port ran a web server. We then launched Nmap against all configured IP addresses and checked which operating systems Nmap identifies. For 555 fingerprints, Nmap uniquely identified the operating system simulated by Honeyd. For 37 fingerprints, Nmap presented a list of possible choices that included the simulated personality. Nmap failed to identify the correct operating system for only 8 fingerprints. This might be a problem of Honeyd, or it could be due to a badly formed fingerprint database.

5 Applications

In this section, we describe how the Honeyd framework can be used in different areas of system security.

```
$ 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 7: Using traceroute, we measure a routing path in the virtual routing topology. The measured latencies match the configured ones.

5.1 Network Decoys

The traditional role of a honeypot is that of a network decoy. Our framework can be used to instrument the unallocated addresses of a production network with virtual honeypots. Adversaries that scan the production network can potentially be confused and deterred by the virtual honeypots. In conjunction with a NIDS, the resulting network traffic may help in getting early warning of attacks.

5.2 Detecting and Countering Worms

Honeypots are ideally suited to intercept traffic from adversaries that randomly scan the network. This is especially true for Internet worms that use some form of random scanning for new targets [23], *e.g.* Blaster [5],

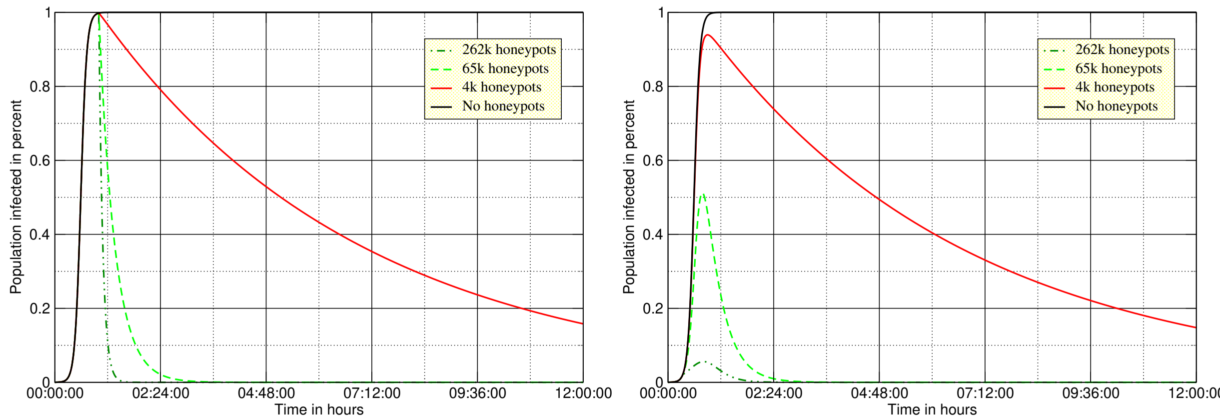

Figure 8: The graphs show the simulated worm propagation when immunizing infected hosts that connect to a virtual honeypot. The left graph shows the propagation if the virtual honeypots are activated one hour after the worm starts spreading. The right graph shows the propagation if the honeypots are activated after 20 minutes.

Code Red [13], Nimda [4], Slammer [14], etc. In this section, we show how a virtual honeypot deployment can be used to detect new worms and how to launch active countermeasures against infected machines once a worm has been identified.

To intercept probes from worms, we instrument virtual honeypots on unallocated network addresses. The probability of receiving a probe depends on the number of infected machines i, the worm propagation chance and the number of deployed honeypots h. The worm propagation chance depends on the worm propagation algorithm, the number of vulnerable hosts and the size of the address space. In general, the larger our honeypot deployment the earlier one of the honeypots receives a worm probe.

To detect new worms, we can use the Honeyd framework in two different ways. We may deploy a large number of virtual honeypots as gateways in front of a smaller number of high-interaction honeypots. Honeyd instruments the virtual honeypots. It forwards only TCP connections that have been established and only UDP packets that carry a payload that fail to match a known fingerprint. In such a setting, Honeyd shields the high-interaction honeypots from uninteresting scanning or backscatter activity. A high-interaction honeypot like ReVirt [6] is used to detect compromises or unusual network activity. Using the automated NIDS signature generation proposed by Kreibich *et al.* [12], we can then block the detected worm or exploit at the network border. The effectiveness of this approach has been analyzed by Moore *et al.* [15]. To improve it, we can configure Honeyd to replay packets to several high-interaction honeypots that run different operating systems and software versions.

On the other hand, we can use Honeyd's subsystem support to expose regular UNIX applications like OpenSSH to worms. This solution is limiting as we are restricted to detecting worms only for the operating system that is running the framework and most worms target Microsoft Windows, not UNIX.

Moore *et al.* show that containing worms is not practical on an Internet scale unless a large fraction of the Internet cooperates in the containment effort [15]. However, with the Honeyd framework, it is possible to actively counter worm propagation by immunizing infected hosts that contact our virtual honeypots. Analogous to Moore *et al.* [15], we can model the effect of immunization on worm propagation by using the classic SIR epidemic model [11]. The model states that the number of newly infected hosts increases linearly with the product of infected hosts, fraction of susceptible hosts and contact rate. The immunization is represented by a decrease in new infections that is linear in the number of infected hosts:

$$\begin{aligned} \frac{ds}{dt} &= -\beta\, i(t)s(t) \\ \frac{di}{dt} &= \beta\, i(t)s(t) - \gamma\, i(t) \\ \frac{dr}{dt} &= \gamma\, i(t), \end{aligned}$$

where at time t, $i(t)$ is the fraction of infected hosts, $s(t)$ the fraction of susceptible hosts and $r(t)$ the fraction of immunized hosts. The propagation speed of the worm is characterized by the contact rate β and the immunization rate is represented by γ.

We simulate worm propagation based on the parameters for a Code-Red like worm [13, 15]. We use 360,000 susceptible machines in a 2^{32} address space and set the initial worm seed to 150 infected machines. Each worm launches 50 probes per second. The simulation measures the effectiveness of using active immunization by virtual honeypots. The honeypots start working after a time delay. The time delay represents the time that is required to detect the worm and install the immunization code. We expect that immunization code can be prepared before a vulnerability is actively exploited. Figure 8 shows the worm propagation resulting from a varying number of instrumented honeypots. The graph on the left shows the results if the honeypots are brought online an hour after the worm started spreading. The graph on the right shows the results if the honeypots can be activated within 20 minutes. If we wait for an hour, all vulnerable machines on the Internet will be infected. Our chances are better if we start the honeypots after 20 minutes. In that case, a deloyment of about 262,000 honeypots is capable of stopping the worm from spreading.

Alternatively, it would be possible to scan the Internet for vulnerable systems and remotely patch them. For ethical reasons, this is probably unfeasible. However, if we can reliably detect an infected machine with our virtual honeypot framework, then active immunization might be an appropriate response. For the Blaster worm, this idea has been realized by Oudot *et al.* [16].

5.3 Spam Prevention

The Honeyd framework can be used to understand how spammers operate and to automate the identification of new spam which can then be submitted to collaborative spam filters.

In general, spammers abuse two Internet services: proxy servers [8] and open mail relays. Open proxies are often used to connect to other proxies or to submit spam email to open mail relays. Spammers can use open proxies to anonymize their identity to prevent tracking the spam back to its origin. An open mail relay accepts email from any sender address to any recipient address. By sending spam email to open mail relays, a spammer causes the mail relay to deliver the spam in his stead.

To understand how spammers operate we use the Honeyd framework to instrument networks with open proxy servers and open mail relays. We make use of Honeyd's GRE tunneling capabilities and tunnel several C-class networks to a central Honeyd host.

We populate our network space with randomly chosen IP addresses and a random selection of services. Some virtual hosts may run an open proxy and others may just run an open mail relay or a combination of both.

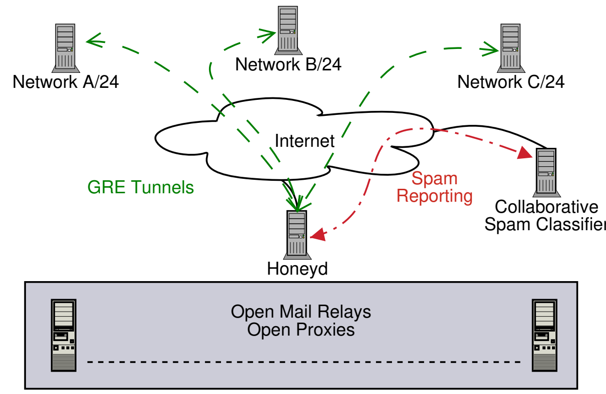

Figure 9: Using the Honeyd framework, it is possible to instrument networks to automatically capture spam and submit it to collaborative filtering systems.

When a spammer attempts to send spam email via an open proxy or an open mail relay, the email is automatically redirected to a spam trap. The spam trap then submits the collected spam to a collaborative spam filter.

At this writing, Honeyd has received and processed more than 26,000 spam emails. A detailed evaluation is the subject of future work.

6 Related Work

There are several areas of research in TCP/IP stack fingerprinting, among them: effective methods to classify 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 [7]. 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. *P0f* [27] is one such tool. 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 [20]. 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 to contain artifacts of the configured operating system.

High-interaction virtual honeypots can be constructed using User Mode Linux (UML) or Vmware [25]. One example is ReVirt which can reconstruct the state of the virtual machine for any point in time [6]. This is very helpful for forensic analysis after the virtual machine has been compromised. Although high-interaction virtual honeypots can be fully compromised, it is not easy to instrument thousands of high-interaction virtual machines due to their overhead. However, the Honeyd framework allows us to instrument unallocated network space with thousands of virtual honeypots. Furthermore, we may use a combination of Honeyd and virtual machines to get the benefit of both approaches. In this case, Honeyd provides network facades and selectively proxies connections to services to backends provided by high-interaction virtual machines.

7 Conclusion

Honeyd is a framework for creating virtual honeypots. Honeyd mimics the network stack behavior of operating systems to deceive fingerprinting tools like Nmap and Xprobe.

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

We showed how the Honeyd framework can be deployed to help in different areas of system security, *e.g.*, worm detection, worm countermeasures, or spam prevention.

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

8 Acknowledgments

I thank Marius Eriksen and Peter Honeyman for careful reviews and suggestions. Jamie Van Randwyk, Dug Song and Eric Thomas also provided helpful suggestions and contributions.

References

[1] Ofir Arkin and Fyodor Yarochkin. Xprobe v2.0: A "Fuzzy" Approach to Remote Active Operating System Fingerprinting. `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] CERT. Cert advisory ca-2001-26 nimda worm. `www.cert.org/advisories/CA-2001-26.html`, September 2001.

[5] CERT. Cert advisory ca-2003-20 w32/blaster worm. `www.cert.org/advisories/CA-2003-20.html`, August 2003.

[6] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, December 2002.

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

[8] S. Glassman. A Caching Relay for the World Wide Web. In *Proceedings of the First International World Wide Web Conference*, pages 69–76, May 1994.

[9] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation (GRE). RFC 1701, October 1994.

[10] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation over IPv4 networks. RFC 1702, October 1994.

[11] Herbert W. Hethcote. The Mathematics of Infectious Diseases. *SIAM Review*, 42(4):599–653, 2000.

[12] C. Kreibich and J. Crowcroft. Automated NIDS Signature Generation using Honeypots. Poster paper, ACM SIGCOMM 2003, August 2003.

[13] D. Moore, C. Shannon, and J. Brown. Code-Red: A Case Study on The Spread and Victims of an Internet Worm. In *Proceedings of the 2nd ACM Internet Measurement Workshop*, pages 273–284. ACM Press, November 2002.

[14] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1(4):33–39, July 2003.

[15] David Moore, Colleen Shannon, Geoffrey Voelker, and Stefan Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proceedings of the 2003 IEEE Infocom Conference*, April 2003.

[16] Laurent Oudot. Fighting worms with honeypots: honeyd vs msblast.exe. `lists.insecure.org/lists/honeypots/2003/Jul-Sep/0071.html`, August 2003. Honeypots mailinglist.

[17] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

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

[19] Thomas Ptacek and Timothy Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Secure Networks Whitepaper, August 1998.

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

[21] Dug Song, Robert Malan, and Robert Stone. A Snapshot of Global Worm Activity. Technical report, Arbor Networks, November 2001.

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

[23] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in your Spare Time. In *Proceedings of the 11th USENIX Secuirty Symposium*, August 2002.

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

[25] Jeremy Sugerman, Ganesh Venkitachalam, , and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the Annual USENIX Technical Conference*, pages 25–30, June 2001.

[26] David Wagner and Paolo Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, November 2002.

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