HoneydV6: A Low-interaction IPv6 Honeypot

Sven Schindler[1], Bettina Schnor[1], Simon Kiertscher[1], Thomas Scheffler[2] and Eldad Zack[3]
[1]*Department of Computer Science, University of Potsdam, Potsdam, Germany*
[2]*Department of Electrical Engineering, Beuth Hochschule, Berlin, Germany*
[3]*EANTC AG, Berlin, Germany*
{*sschindl, schnor, kiertscher*}*@cs.uni-potsdam.de, scheffler@beuth-hochschule.de, zack@eantc.de*

Keywords: IPv6, Honeypot, Darknet, IPv6 Traffic Analysis.

Abstract: This paper starts with the presentation of results from an IPv6-darknet experiment that we conducted during summer 2012. The experiment indicates that attackers are gaining interest in IPv6 networks and appropriate security tools need to be readied. Therefore, we propose *HoneydV6*, a low-interaction IPv6 honeypot that can simulate entire IPv6 networks and which may be utilized to detect and analyze IPv6 network attacks. Our implementation extends the well-known low-interaction honeypot *Honeyd*. To the best of our knowledge, this is the first low-interaction honeypot which is able to simulate entire IPv6 networks on a single host. The huge IPv6 address spaces requires new approaches and concepts in order to enable attackers to find and exploit a honeypot. We increase the chance for an attacker to find a target host in our IPv6 honeypot by reacting to the attacker's requests with the dynamic generation of new IPv6 host instances in the honeynet.

1 INTRODUCTION

In June 2012, the Internet Society arranged the World IPv6 Launch Day, an event where well-known service providers and web companies like Google or Yahoo! started to enable IPv6 support for their customers.

With the increasing number of service providers offering IPv6, the number of attackers aiming for these networks may increase. In order to get an idea of the current threat level in IPv6 networks, we started an IPv6-darknet experiment in March 2012 using a /48 network. A darknet is an address space that is advertised and routed but does not provide any services (Ford et al., 2006). All traffic entering a darknet can be considered malicious. This eases classification and subsequent analysis, because we do not have to separate production traffic from attack traffic.

Due to the huge IPv6 address space, brute-force network scanning of IPv6 addresses is not attractive for an attacker and hence the probability to catch attackers in a darknet is low. Nevertheless, the results of our darknet experiment show that malicious IPv6 traffic is existent and increasing.

There may also arise new threats that are aimed at specific weaknesses in the IPv6 design. A well-known example for this trend is the published THC-IPv6 Attack Toolkit (Heuse, nd) which exploits several protocol-specific features, such as the IPv6 Stateless Address Autoconfiguration (Thomson et al., 2007).

In order to analyse IPv6-related attacks, IPv6-enabled security tools like Intrusion Detection Systems or virtual honeypots have to be deployed that allow a deeper analysis of attack patterns. Virtual honeypots provide an excellent mechanism to collect information about network attacks and vulnerabilities, because they provide a level of interactivity that cannot be achieved by darknets. A virtual honeypot is a security device that has no production value (Seifert et al., 2006). This can be something like a computer or even a mobile phone which only purpose is to attract attackers, so that their attacks can be analysed. Low-interaction honeypots like *Honeyd* (Provos, 2003) may even be used to simulate large networks with thousands of routers and hosts.

We chose to extend the low-interaction IPv4 honeypot *Honeyd* to *HoneydV6*, since it is able to simulate entire IPv4 networks on a single computer and provides a lot of components that could be reused in our IPv6 implementation. Further, *Honeyd* is the fundamental part of a number of honeypot solutions like *Tiny Honeypot* or the *SCADA HoneyNet Project* and can be used to improve the capabilities of honeypots like Nephentes (CERT Polska, 2012). Therefore, by implementing IPv6 functionality into *Honeyd*, the aforementioned honeypots may be adapted so that they are able to handle IPv6 connections as well.

Honeyd implements a customized network stack to handle multiple simulated hosts on a single machine. The large number of available addresses in a single IPv6 network requires a new honeynet design approach. The static placement of virtual components does not work, because an attacker is unlikely to find a deployed honeypot within the large IPv6 address space by chance. We therefore developed the concept of *random IPv6 request processing* to allow attackers to dynamically find and exploit our simulated hosts.

The next section presents the results from our darknet experiment. Section 3 summarizes related work. In Section 4 and 5, we present the IPv6 extension of *Honeyd*. Section 6 shows the results of performance measurements of *HoneydV6* and Section 7 concludes our work.

2 EXAMINING THE THREAT LEVEL IN IPv6 NETWORKS

A couple of years ago, it was hard to find any malicious or even unintentional traffic in IPv6 networks.

In 2006, Matthew Ford et al. published a traffic statistic of their IPv6 darknet with a /48 prefix, which may have been the world's first IPv6 darknet (Ford et al., 2006). Within approximately 16 months, they captured about 12 ICMPv6 packets which were most probably caused by misconfiguration and typographical errors resulting from the long and unwieldy IPv6 addresses. In comparison, Pang et al. observed in 2004 about 30,000 packets of background radiation per second in a class A IPv4 network (Pang et al., 2004).

In 2010, Geoff Huston presented the results of his darknet experiment where he examined the background radiation in a 2400::/12 network provided by APNIC for 9 days (Huston, 2010). The darknet received about 21,000 packets. However, the used /12 address block was not vacant and about 1.6 percent of the network addresses had already been allocated. Therefore, it is hard to compare the results of this experiment to earlier darknet results even though traffic which was directed to allocated addresses was filtered before further analysis. It is assumed that the received traffic is caused by misconfiguration and probably a small number of guess probes. Scans that are definitely produced by bots or viruses could not be detected.

We set up a new /48 IPv6 darknet and monitored the incoming traffic for 9 months including the time around the World IPv6 Launch to confirm this assumption. The address space was provided by the tunnel broker “Hurricane Electric” and the incoming traffic was tunnelled to our machine using a SIT tunnel.

While the probability that an attacker choses an IPv6 address from our darknet is about 2^{-48}, we observed a total number of 1172 packets. The whole traffic consists of TCP packets, much to our surprise, we didn't receive a single UDP or ICMPv6 packet. Figure 1 shows the temporal distribution of the received packets. As predicted, we received most of the traffic around the World IPv6 Launch day. Even though the number of received packets has decreased since the World IPv6 Launch, we are still constantly receiving packets.

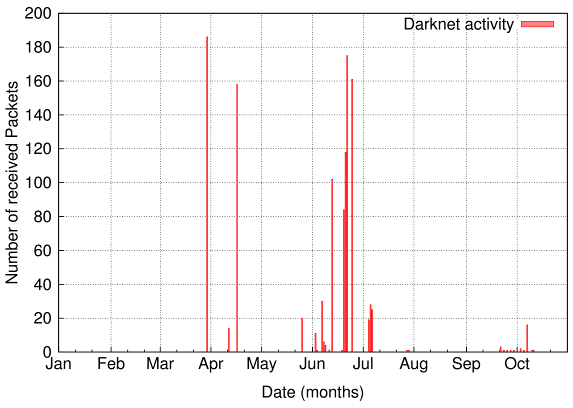

Figure 1: Number of received packets per day, increased number of packets around the World IPv6 Launch day.

2.1 Backscatter

Most of the TCP traffic (1157 packets) seems to be backscatter. This kind of traffic can be caused by misconfiguration or by attackers who intentionally use spoofed source addresses when sending packets to a destination. The destination under attack creates a respond packet to the spoofed source address. So in our case, attackers spoofed addresses that belong to our darknet address space.

In case of TCP it is rather simple to spot backscatter traffic. A TCP handshake is essential to enable the connection setup. Normally, this handshake cannot be completed if the initiating client uses a spoofed source address. If a target receives the initial TCP handshake packet, where the SYN flag is set in the TCP header, it tries to complete the handshake by answering with a TCP packet where SYN and ACK flags are set. Hence, the reception of TCP darknet traffic, where SYN and ACK flags are set, is a good indication of backscatter. Of course, it is possible to generate TCP packets with SYN and ACK flags set and send it directly to a destination. However, we concluded that the intentional forwarding of such packets to our darknet is very unlikely, since they would serve no known purpose.

We continued our attack evaluation with an analysis of the attacked ports. Table 1 provides a source port statistic for the received backscatter traffic.

Table 1: Source ports of the received backscatter packets.

Number of packets	Source port	Description
486	113	auth
327	22	ssh
186	6667	ircd
158	80	http

Port 113 belongs to the most occurring source ports in our backscatter traffic. It is actually used by the Ident protocol (Johns, 1993) which is able to identify an owner of a TCP connection on a remote multi-user system. The protocol is still used, for example by IRC servers which connect back via Ident to a requesting source in order to ensure a user's identity (Oikarinen and Reed, 1993).

The 486 received packets with source port 113 came from 8 different source IPs. They are aimed for 457 different destination IPs. This indicates that 457 different clients tried to connect to 8 different servers. A peculiar aspect of all packets received on port 113 is an unaltered acknowledgment number for most of the sources with different destination addresses. Hence, the TCP handshake must have always been initiated with the same initial sequence number with different source addresses. In some cases, even the sequence number as well as acknowledgment number stay unaltered.

As you can see in Table 1, we received 327 packets from source port 22 (ssh). The packets came from 8 different sources and were targeted at 295 different destinations. Two of the source addresses are also contained in the set of packets coming from port 113. Similar to the packets coming from port 113, most packets from the same source share the same acknowledgment number even though the targets are different.

Furthermore, we received 186 packets targeted at port 6667, commonly used by IRC (Kalt, 2000). All packets came from the same source but had a different destination addresses. The acknowledgment number of all packets is equal.

We received further 158 packets coming from one source IP using source port 80. Like the packets coming from port 6667, the acknowledgment number and the target port always stays the same, with one exception. The last packet received contains a different destination port and a different acknowledgment number.

Geoff Huston also reported a huge amount of TCP backscatter traffic where ACK and SYN flags are set. He assumes misconfiguration as one possible explanation for receiving these packets in his darknet. In our case, almost all packets coming from the same source, even packets with different target addresses, share the same target port and acknowledgment number. This indicates a deliberate use of spoofed source addresses when connecting to the server. It is possible that these packets belong to a denial of service attack. Because we might have seen only a subset of all packets belonging to an attack, we are not able to provide a clearer statement about the attack's purpose.

2.2 ACK Scans

We also received 15 packets where only the ACK flag of the TCP header is set without any sign of a prior TCP handshake. All 15 packets are coming from the same /64 subnet, which belongs to the address space of the tunnel broker "Hurricane Electric". The missing handshake suggests that these packets are part of an ACK scan, which is usually used to evaluate filter rules of firewalls. The source port of these packets, however, is Microsoft's file sharing port 445, which belongs to the most attacked ports in the IPv4 darknet experiment presented in (Pang et al., 2004). Geoff Huston also received 141 TCP packets without an initial handshake and he also concludes that these packets belong to a network probe and rules out that these packets may belong to backscatter traffic.

2.3 Summary

Even though our /48 IPv6 darknet recorded only light traffic, we can say that the IPv6 network is not free of threats anymore. Almost all received packets were caused by spoofed source addresses and may belong to denial of service attacks and we even received packets that may be network probes. So far, we did not receive any connection attempts that may be attributed to viruses or bots.

We conclude that, in contrast to earlier darknet reports, the IPv6 internet has become more interesting for attackers.

Our IPv6 darknet has been an excellent tool to assess the general network threat-level, however, is not well suited to analyse network-level attacks in more detail. Therefore, the next section looks at IPv6 honeypots.

3 RELATED WORK

The only IPv6-capable general purpose low-interaction honeypot is Dionaea (Dionaea, nd), a honeypot which emulates well-known services like SMB or SIP. It is able to detect remote shellcode attacks using the emulation library *libemu* (Baecher and Koetter, nd). In contrast to *Honeyd*, Dionaea does not implement a customized network stack and a single instance of Dionaea is not able to simulate entire IPv6 networks. Although it is possible to create honeynets by setting up multiple instances of Dionaea, it is more challenging to maintain multiple machines and expenses increase as additional performance is needed. This approach is not usefully applicable for IPv6 networks if a huge number of honeypots needs to be deployed.

There already exist different approaches to set up *honeyfarms* consisting of thousands of honeypots: In (Vrable et al., 2005), the authors presented *Potemkin*, an architecture to create a honeyfarm with thousands of virtual machine based honeypots. A gateway dynamically creates virtual machines for incoming requests and forwards the traffic to the machines. It is not clear whether the gateway is able to process IPv6 traffic. The approach needs to filter network scans because otherwise, a new machine would have to be created for each scanned IP address which would lead to performance problems. Of course, Potemkin faces the same performance issues as most high-interaction honeypots do. While Potemkin needs a handful of servers to simulate 64,000 machines, low-interaction honeypots like *Honeyd* are able to simulate the same number of hosts on a single end user machine.

Recently, *HoneyCloud* was proposed (Clemente et al., 2012), a cloud based honeypot that aims to be able to handle thousands of attackers and to utilize various log mechanisms and IDS'. HoneyCloud creates new virtual machine based high-interaction honeypots for each attacker and is deployed in an elastic compute cloud (EC2) using Eucalyptus. The system utilizes different log mechanisms and is even able to capture keystrokes. While the Potemkin honeyfarm may assign multiple attackers to the same target machine, HoneyCloud assigns each attacker to a separate high-interaction honeypot which writes events into own log files in order to avoid log file mixtures. HoneyCloud accepts SSH connections only and is currently not able to handle other services or even network scans. That is a drawback when trying to gather valuable information about bots and viruses in IPv6 networks because it is necessary to monitor the whole range of ports and services. Furthermore, the need for a cloud infrastructure makes it hard for smaller businesses or even private researchers to deploy the honeypot without falling back on commercial solutions.

4 EXTENDING Honeyd TO Honeyd V6

Honeyd is a low-interaction honeypot which has been developed by Niels Provos in the C programming language and is currently available in version 1.5c on the project website[1]. We chose *Honeyd* as base for our IPv6 honeypot since it is able to simulate entire IPv4 networks on a single host. It provides a framework that enables users to write service scripts for the simulated machines, e.g. a script that simulates a telnet service and captures all log-in attempts of an attacker. These service scripts can be bound to addresses which are managed by *Honeyd*.

The simulation of entire networks in *Honeyd* is accomplished by a customized network stack implementation using the network capture library *libpcap*[2] to bypass the host's network stack. Even though this approach is very flexible, it impedes the IPv6 extension because the existing IPv6 functionality of the host's operating system cannot be reused. The packet processing has to be modified and essential parts of entirely new protocols such as ICMPv6 or the Neighbor Discovery Protocol (NDP) have to be implemented.

In this section, we will describe the major IPv6 specific implementations. A number modifications require a deeper understanding of *Honeyd's* architecture. We will therefore provide a deeper insight into the technical background when required.

4.1 Adapting the Configuration of Virtual Hosts

Honeyd can be configured by defining all hosts to be simulated in a configuration file. The behaviour of a simulated host can be specified via so-called system templates. A template specifies system properties such as open ports and their assigned scripts. Listing 1 shows a configuration file for an IPv4 network containing two system templates called *windows* and *linux*.

```
create windows
set windows default tcp action reset
add windows tcp port 21 "scripts/ftp.sh"


create linux
set linux default tcp action reset
add linux tcp port 23 "scripts/telnet.pl"
add linux tcp port 80 "scripts/web.sh"
```

[1]http://www.honeyd.org/
[2]http://www.tcpdump.org/

```
set windows ethernet "aa:00:04:78:98:76"
set linux ethernet "aa:00:04:78:95:82"

bind 192.168.1.5 windows
bind 192.168.1.6 windows
bind 192.168.1.7 linux
```

Listing 1: Honeyd example configuration.

A template is created using the *create* statement followed by the template name. In this example, the FTP port 21 of the *windows* template is opened and attached to a script called *ftp.sh*. The *ftp.sh* script contains just enough functionality to capture all log-in attempts, an actual log-in is not possible. The *set* statement assigns a MAC address to the template. By using the *bind* statement, the *windows* template is bound to the addresses *192.168.1.5* and *192.168.1.6* whereas the *linux* template is bound to the address *192.168.1.7*.

Internally, *Honeyd* creates a new template for each IP address binding which are basically copies of the original defined template. The names of the copied templates are changed from *windows* or *linux* to their defined IP addresses so that a template belonging to an incoming connection can easily be found by its name.

The different templates are maintained in a splay tree ordered by their names. A splay tree is a self balancing binary tree where recently accessed elements are located close to the root (Sleator and Tarjan, 1985). This allows an efficient search for a connection belonging to an incoming packet.

In *HoneydV6*, the syntax to define templates and to assign scripts to configured ports in the configuration file is left unchanged. Our modified configuration parser allows users to bind templates to an IPv6 address in the same way as an IPv4 address. A *bind* statement with a given IPv6 address followed by the template name is sufficient to bind a template to an IPv6 address.

The fact that the honeypot maintains templates in a splay tree ordered by their names in a string representation allows us to store IPv6 and IPv4 templates in the same tree. It might be possible to improve the performance by storing IPv4 and IPv6 templates in two separate trees. However, our performance tests show that the current performance is sufficient for most scenarios (see Section 7).

4.2 Modifying Packet Processing

As soon as *Honeyd* receives an IPv4 packet, it searches for the corresponding template based on the target address. If it cannot find a template, the packet will silently be discarded. If a packet is received for which *Honeyd* is responsible, the packet will be forwarded to a dispatcher. The dispatcher moves the packet further to a TCP, UDP or ICMP processor, depending on the IP payload. If the packet is a fragment, then *Honeyd* will wait for all fragments to arrive and will assemble the fragment before forwarding it to the dispatcher.

The service scripts, such as the *ftp.sh* script of the previous example, are connected to the matching connection via socket pairs. *Honeyd* forwards incoming traffic to the standard input of the assigned script while the standard output of a script is sent back to the attacker. In addition, scripts are able to print logging information using their standard error output.

Similar to the IPv4 approach, *HoneydV6* assembles and forwards incoming IPv6 packets to a new IPv6 packet dispatcher. We had to modify the original TCP and UDP processor, so that they are able to process both kinds of connections, IPv4 as well as IPv6. The IPv6 dispatcher forwards received packets to the new ICMPv6 or to the extended TCP and UDP processor based on the payload type.

Fragmented IPv6 packets get reassembled before they are forwarded to the IPv6 packet dispatcher. This function required the implementation of an IPv6 packet assembler which evaluates the fragment extension header, if available, of each incoming packet. The offset and length of each incoming fragment is logged so that attacks which are based on packet fragmentation can easily be analysed.

Honeyd provides a number of further settings and mechanisms such as proxy connections to high-interaction honeypots, conditional templates and fingerprinting. However, these features are out of the scope of this document.

4.3 TCP and UDP

Honeyd's packet dispatcher passes incoming TCP and UDP packets to the corresponding callbacks. These callback functions are named *tcp_recv_cb* and *udp_recv_cb* respectively. After our modifications, these functions wrap around *tcp_recv_cb46* and *udp_recv_cb46* which are able to handle IPv4 as well as IPv6 packets.

Fortunately, these callbacks needed only minor modifications. Depending on the address family, an incoming packet is now mapped to the corresponding structure as shown in the following code snippet of the UDP callback:

```
if (addr_family == AF_INET) {
  ip = (struct ip_hdr *)pkt;
  udp = (struct udp_hdr *)(pkt + (ip->ip_hl << 2));

}else if (addr_family == AF_INET6) {
  ip6 = (struct ip6_hdr *)pkt;
  get_ip6_next_hdr((u_char **)&udp,ip6,IP_PROTO_UDP);
}
```

Listing 2: Protocol switches to handle IPv4 and IPv6.

The use of the two different structures in the same functions had quite an impact on multiple code segments. However, this way a lot of code fragmentation could be avoided and the packet processing is easier to understand.

In quite a few sections of the TCP and UDP code, the IPv4 functionality could not be reused and a protocol switch had to be implemented. One example is the checksum and data length calculation, which had to be updated in both callbacks.

The IPv6 packet processing needs to be aware of possible extension headers. As shown in the previous example, the actual payload cannot be retrieved directly, we first have to parse the chain of possible extension headers. The function *get_ip6_next_hdr* provides a pointer to a certain extension header or the actual payload.

The structures to maintain UDP and TCP connections (*udp_con* and *tcp_con*) include a pointer to a tuple structure which holds address details of a connection:

```
struct tuple {
 ...
  //currently used to store the ipv4 addresses
  ip_addr_t ip_src;
  ip_addr_t ip_dst;
  //currently used to store the ipv6 addresses
  struct addr src_addr;
  struct addr dst_addr;
  uint16_t sport;
  uint16_t dport;
 ...
};
```

Listing 3: Excerpt of the modified tuple structure to maintain a connection.

Honeyd uses the variables *ip_src* and *ip_dst* of type *ip_addr_t* to store IPv4 addresses of a connection. This type is too small to store IPv6 addresses, so we had to add the fields *src_addr* and *dst_addr* to store IPv6 addresses.

4.4 Fragmentation

IPv6 fragmentation handling differs from IPv4 insofar, as only source nodes may fragment packets. We implemented the functions *ip6_send_fragments* and *ip6_fragment* that handle fragmentation of outgoing IPv6 packets that are larger than the maximum transmission unit (MTU) and reassemble fragmented incoming packets. All fragments are maintained in a splay tree using the following *fragment6* structure:

```
struct fragment6 {
  SPLAY_ENTRY(fragment6) node;
  TAILQ_ENTRY(fragment6) next;
  TAILQ_HEAD(frag6q, frag6ent) fraglist;

  struct addr src_addr;
  struct addr dst_addr;

  uint32_t ip6_id;
  uint32_t total_len;
  uint8_t nxt_hdr;
  struct event timeout;
};
```

Listing 4: IPv6 fragment structure.

Besides address, length and ID, the structure contains a queue which stores received fragments belonging to a packet. When a packet arrives, the function *ip6_fragment_find* is used to search for already received fragments in the splay tree. If the received packet is the first received fragment then *ip6_fragment_new* is used to insert a new entry into the splay tree. If other fragments have already been received, then *ip6_insert_fragment* is used to add the packet to the fragment queue.

Outgoing packets bigger than the Honeyd MTU are fragmented using *ip6_send_fragments*. Path MTU discovery has not yet been implemented and a fixed defined size *HONEYD_MTU* is used instead. *ip6_send_fragments* computes the number of fragments needed and prepares the fragments by inserting a fragmentation extension header before using *honey_deliver_ethernet6* to send each single fragment.

4.5 Implementation of the Neighbor Discovery Protocol

While IPv4 uses ARP for address resolution, IPv6 is based on the new so-called Neighbor Discovery Protocol (NDP). Therefore, *HoneydV6* has to implement the essential parts of NDP.

For every IPv4 template that is created, *Honeyd* creates an ARP entry which contains the Ethernet address in a splay tree that can be used later to handle ARP requests. For IPv6 templates, *HoneydV6* creates a further splay tree representing a neighbor cache. It contains the Ethernet addresses of all IPv6 templates needed by the NDP.

We implemented the essential parts of NDP that are required to properly advertise the simulated machines in the network:

- Send and Process Neighbor Solicitations - If a machine needs the Ethernet address of a node in the local network, it sends a neighbor solicitation message to that node. A host receiving a neighbor solicitation answers with a neighbor advertisement containing the corresponding Ethernet address.

- Send Router Solicitations and Process Router advertisements - It is very probable that in practice *HoneydV6* will run behind a router. In order to find all routers and their Ethernet addresses, *HoneydV6* sends a router solicitation to the all routers multicast address and afterwards collects incoming router advertisements.

Because NDP goes hand in hand with ICMPv6, the core functionality to handle NDP packets is contained in *icmp6.c*. Honeyd's dispatcher was modified to forward ICMPv6 packets to the ICMPv6 dispatcher function *icmp6_recv_cb*. The function passes the incoming packet to the corresponding handler depending on the ICMPv6/NDP type.

```
switch(icmp6->icmp6_type){
 case ND_NEIGHBOR_SOLICIT:
   handle_neighbor_solicitation(inter,ip6, icmp6);
   break;
 case ND_NEIGHBOR_ADVERT:
   handle_neighbor_advertisement(inter,ip6, icmp6);
   break;
 case ND_ROUTER_ADVERT:
   handle_router_advertisement(inter,ip6 ,icmp6);
   break;
 case ICMP6_ECHO_REQUEST:
   handle_echo_request(inter,ip6, icmp6,
     ntohs(ip6->ip6_plen)+IP6_HDR_LEN+ETH_HDR_LEN
   break;
 default:
 syslog(LOG_DEBUG,"unhandled icmp6 type: %d",
     icmp6->icmp6_type);
   break;
}
```

Listing 5: ICMPv6 dispatcher.

4.6 Support for the Monitoring of Network Scans

One of *Honeyd's* advantages is its ability to simulate entire network topologies containing virtual routers and virtual low-interaction hosts. This mechanism allows researchers to analyse the way network scans are performed and how bots try to find new hosts to infect. RFC 5157 (Chown, 2008) suggests a number of possible ways to reveal IPv6 hosts more efficiently than brute-force network scanning. Network scanning tools like *scan6* of the *SI6 Networks' IPv6 Toolkit* (SI6 Networks, 2012) already started to implement these scanning techniques.

In order to allow researchers to observe new kinds of scanning methods in IPv6 networks, we adapted the internal routing mechanisms of *Honeyd* to support IPv6 packet routing.

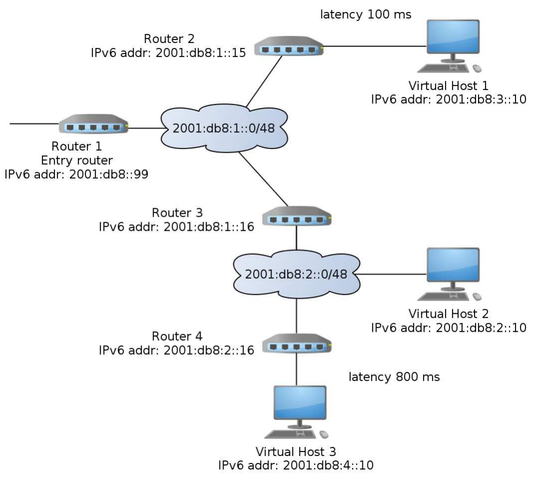

Figure 2: An example IPv6 network that can be simulated using *HoneydV6* and the configuration presented in Listing 6.

Listing 6 shows an example configuration for the network topology presented in Figure 2. In order to simplify the configuration, the configuration syntax corresponds with the syntax used to define IPv4 network topologies. Our example contains four virtual routers and three virtual low-interaction hosts. Incoming network packets need to traverse an entry router, which in this example has the IPv6 address 2001:db8::99. An entry router can be defined using the *route entry* statement followed by the router address and the reachable network which in this case is 2001:db8::0/32.

By using the *add net* and the *link* statement, the entry router is directly connected to *Router 2* and *Router 3* with the addresses 2001:db8:1::15 and 2001:db8:1::16 respectively. *Router 2* covers the network 2001:db8:3::/48 and has the virtual low-interaction *Host 1* with address 2001:db8:3::10 attached.

Because of the first *add net* statement, *HoneydV6* knows that packets targeting the network 2001:d8:3::/48 need to be forwarded to *Router 2*. A *link* statement defines what addresses are directly reachable through a router. In case of *Router 2*, all addresses within the network 2001:db8:3::/48 are directly reachable which includes *Host 1*.

In order to simulate a realistic network packet routing, the following ICMPv6 types had to be implemented:

- Time Exceeded - Each time an IPv6 packet traverses a router, its hop limit gets decreased. As soon as the hop limit reaches zero, *HoneydV6* sends an ICMPv6 *Time Exceeded* message back to the source.

- Destination Unreachable - If a packet is sent to an address within *Honeyd's* address space to an undefined virtual host or to a closed UDP port then the honeypot replies with an ICMPv6 *Destination Unreachable* message.

Both packet types are essential in order to make network scanning tools like *traceroute6* work and to allow attackers exploring the virtual network.

The simulation of physical network properties, as provided by the *Honeyd* IPv4 version, was adapted to also work with IPv6 packets. This includes the computation of the hop limit of a packet and functions that find and compare IPv6 networks.

It is possible to define factors like packet loss or network latency as shown in Listing 6. In this example, a packet transfer from *Router 1* to *Virtual Host 1* takes about 100 milliseconds while a packet from *Router 3* to *Virtual Host 3* needs about 800 milliseconds. If no latency is set, then a packet is passed to the next hop without any extra delay except the time needed for computation.

Of course, the provided example configuration requires that the covered prefixes are advertised throughout the global IPv6 Internet and attacking traffic is forwarded to the machine *HoneydV6* is running on.

```
route entry 2001:db8::99 network 2001:db8::0/32

bind 2001:db8::99 router1
bind 2001:db8:1::15 router2
bind 2001:db8:1::16 router3
bind 2001:db8:2::16 router4

bind 2001:db8:3::10 host1
bind 2001:db8:2::10 host2
bind 2001:db8:4::10 host3


route 2001:db8::99
  add net 2001:db8:3::0/48 2001:db8:1::15
  latency 100 ms


route 2001:db8::99
  add net 2001:db8:2::0/48 2001:db8:1::16

route 2001:db8::99
  add net 2001:db8:4::0/48 2001:db8:1::16

route 2001:db8:1::16
  add net 2001:db8:4::0/48 2001:db8:2::16
  latency 800 ms

route 2001:db8::99 link 2001:db8:1::0/48
route 2001:db8:1::15 link 2001:db8:3::0/48
route 2001:db8:1::16 link 2001:db8:2::0/48
route 2001:db8:2::16 link 2001:db8:4::0/48
```

Listing 6: Extract of HoneydV6 configuration to simulate the network shown in Figure 2.

5 PITFALLS

We faced two major issues when we extended *Honeyd* to *HoneydV6*. One problem was that scope IDs, which were embedded in link-local addresses, complicated address comparisons needed to route packets. Besides that, we had to deal with memory access violations caused by dynamic arrays. The following two subsections explain both issues in more detail.

5.1 Scope IDs Stored in Link-local Addresses

The link-local interface addresses that we retrieved using the *libdnet* network library function *intf_get* contained scope IDs directly embedded in the address. In order to convert these addresses into valid link-local addresses, the scope IDs had to be removed. We wrote a simple function called *addr_remove_scope_id* to remove the scope ID from link-local addresses.

```
static void addr_remove_scope_id(struct addr* ip6) {
  if (ip6->addr_data8[0]==0xfe && ip6->addr_data8[1]==0x80) {
    /* delete scope id */
    ip6->addr_data8[2]=0;
    ip6->addr_data8[3]=0;
  }
}
```

Listing 7: Function to remove scope IDs.

HoneydV6 retrieves the interface of an incoming packet by using *libpcap*. Therefore there is no need to store a removed scope ID. When *HoneydV6* initializes and inspects an interface, it removes scope IDs of all it's IPv6 address aliases directly after acquiring the interface information with *intf_get*.

```
for(i=0;i<inter->if_ent.intf_alias_num;i++){
  if (inter->if_ent.intf_alias_addrs[i].addr_type == ADDR_TYPE_IP6){
    /* clear the embedded scope id if its a link-local address */
    ip6addr = &inter->if_ent.intf_alias_addrs[i];
    addr_remove_scope_id(ip6addr);
  }
}
```

Listing 8: Removing the scope IDs of all link-local alias addresses.

5.2 Use of Dynamic Arrays

The original *Honeyd* version maintains information about an interface in a custom *interface* structure shown in Listing 9. This structure has a field of type *intf_entry* followed by other fields.

```
struct interface {
  TAILQ_ENTRY(interface) next;

  struct intf_entry if_ent;
  int if_addrbits;
  struct event if_recvev;
  pcap_t *if_pcap;
  eth_t *if_eth;
  int if_dloff;

  char if_filter[1024];
};
```

Listing 9: Structure used to store interface information.

The *intf_entry* structure contains a dynamic array which may overwrite the following fields. The function *intf_get*, which is used in *interface.c* to retrieve interface information, fills the dynamic array with address aliases depending on the amount of reserved memory. If no further memory is available then no alias will be returned. This was not a problem in the IPv4 version because no address aliases needed to be requested. In the IPv6 version, we need to find out the address aliases to get information about assigned IPv6 addresses too. Therefore, we extended the memory allocation for the interface and moved the *intf_entry* structure to the end of the interface structure.

6 COVERING HUGE ADDRESS SPACES USING RANDOM IPv6 REQUEST PROCESSING

The huge address space of an IPv6 subnet makes it hard, if not almost impossible, for an attacker to find a single host on the network by pure chance. While this fact is very welcome in common networks, it impedes the behavioral analysis of an actual attacker who may or may not be able to find a machine.

We want to observe IPv6 network scan techniques and analyse the attacker's actions when he actually finds a running host. In order to accomplish this, we extended *HoneydV6* with a mechanism that dynamically creates simulated hosts on-demand and randomly accepts IPv6 connections. Hence, after a certain number of connection attempts, an attacker will definitely find a machine to exploit.

Furthermore, all connection attempts are logged, even to IPv6 addresses that are not defined in the configuration file. It allows us to analyze IPv6 network scans and to find new scan patterns.

When a packet arrives, *HoneydV6* tries to find the matching virtual low-interaction host. If no host can be found, then a new template will be dynamically created with a specified *acceptance probability*. A user can enable the so-called *IPv6 random mode* by using the *randomipv6* statement followed by the acceptance probability. In order to define what template to use for dynamically created machines, the name of a default template has to be specified right after the acceptance probability.

Consider the example configuration in Listing 10 where we define the template *randomdefault* to be the default template. The default template has the web server and the FTP port open and assigned to the corresponding scripts. Besides the configured open ports and the matching script assignments, the template has a defined Ethernet address. *HoneydV6* replaces the last three bytes of this Ethernet address with randomly generated bytes for each newly created template. This corresponds to *Honeyd's* default behavior in the IPv4 version. Currently, we are supporting only one default template.

```
create randomdefault
set randomdefault default tcp action reset
add randomdefault tcp port 21 "scripts/ftp.sh"
add randomdefault tcp port 80 "scripts/web.sh"
set randomdefault ethernet "aa:00:04:78:98:78"

randomipv6 0.5 randomdefault 256

randomexclude 2001:db8::1
randomexclude 2001:db8::2
randomexclude 2001:db8::3
```

Listing 10: Honeyd configuration to randomly accept IPv6 connections.

If the honeypot randomly decides to reject a request and not to create a machine for it, then the target address will be blacklisted. Future requests to a blacklisted address will always be ignored to keep the system state consistent and to avoid revealing the honeypot.

In some cases it may be useful to exclude certain addresses from the automatic template creation, e.g. if other nodes are in the same network. This can be done by using the *randomexclude* statement. An excluded address is automatically blacklisted and *HoneydV6* will ignore requests to this address.

It is possible to define an upper bound for the number of dynamically created templates by the honeypot. This number can be set after the default template name. In the example above, the maximum number of allowed templates is 256. It is important to restrict the number of dynamically created virtual low-interaction hosts in order to avoid memory-exhaustion attacks. Each created machine and each blacklisted address causes memory consumption until the maximum number of allowed machines is reached.

We recommend to restrict the number of dynamically created machines as well as the acceptance probability to an appropriate low value depending on the use case. A large number of uniformly distributed host may easily reveal the honeypot.

7 PERFORMANCE TESTS

Our modified version of *Honeyd* still fully contains the original IPv4 implementation. Thus it is able to handle IPv4 and IPv6 packets at the same time. When we implemented the IPv6 functionality, we tried to modify the IPv4 code as little as possible in order to avoid new programming errors and negative impact on the IPv4 performance. Nevertheless, in some cases minor modifications to the IPv4 work flow had to be done. We conducted some measurements to quantify the performance of the new IPv4 and IPv6 code in *HoneydV6*.

7.1 Comparison: IPv4 and IPv6 Throughput

In order to evaluate the performance impact of our IPv6 modification, we compared the average application layer throughput of the original IPv4 *Honeyd* 1.5c with *HoneydV6*. We developed a simple *Honeyd* benchmark service script and a corresponding client which allow us to measure the time needed to transfer larger files over the network to the honeypot. The original honeypot as well as the IPv6 modification were installed on a Fujitsu PRIMERGY TX200 S5 Server with an Intel Xeon processor 5500 series and 4096 MB of RAM running Ubuntu 12.04. The benchmark client was installed on a Lenovo ThinkPad L520 with an Intel i5-2450M CPU and 4096 MB of RAM. Both computers were connected via a Brocade FWS648G FastIron switch using Gigabit Ethernet.

Table 2 shows the results for transferring 50 MB and 100 MB from the client via IPv4 and IPv6 to the honeypot benchmark service.

Table 2: Comparison of transmission time in seconds between the original Honeyd version 1.5c and HoneydV6.

Filesize	1.5c (IPv4)	V6 (IPv4)	V6 (IPv6)
50 MB	15.98 s	16.19 s	16.33 s
100 MB	31.85 s	31.94 s	32.36 s

For each experiment, Table 2 shows the median from 5 runs. It takes about 16 seconds to transfer 50 MB to the honeypots and about twice as much time to transfer 100 MB. In case of transferring 50 MB over IPv4, the original 1.5c version of *Honeyd* is approximately 0.2 seconds faster than *HoneydV6*. For sending 100 MB, the original Version was about 0.09 seconds faster than our modified version. This indicates that the overhead is in the magnitude of the measurement error and neglectable. The overhead is most probably caused by a number of newly added IPv4/IPv6 switches in the source code. Furthermore, the IPv6 transfer is insignificantly slower than the IPv4 transfer of both versions. *HoneydV6* needed approximately 0.35 seconds longer than the original 1.5c version to transfer 50 MB and about 0.51 seconds to transfer 100 MB over IPv6.

7.2 Scalability of HoneydV6

While throughput measurements can help to get an impression of the performance impact caused by the IPv6 modifications, throughput is not a very useful criteria to evaluate a honeypot for its suitability in a network. A honeypot like *Honeyd* rather needs to be able to handle a large number of connections than transferring huge files.

Provos and Holz measured for example the number of TCP requests per second that *Honeyd* is able to process (Provos and Holz, 2008). Since we are particularly interested in the performance impact on the application layer, we used the web server benchmark *servload* (Zinke et al., 2012)[3] to measure the number of HTTP GET requests that *HoneydV6* is able to process per second. Servload is capable of replaying a previously captured traffic log file based on the timestamps of the contained packets. We generated a log file containing 20,000 HTTP GET requests from different source addresses with 600 requests per second. *HoneydV6* was configured to simulate a single machine which was bound to an IPv4 and an IPv6 address and which delivers the *web.sh* script that is shipped with the original Honeyd version 1.5c when getting requests on port 80. The *web.sh* script simulates a Microsoft IIS 5.0 and delivers either a directory listing of the server or a 404 NOT FOUND page. Our generated requests demanded a non-existing *index.html* page so that the *web.sh* script responses with an HTTP 404 NOT FOUND error code and a short explanation.

As with the throughput measurements, we repeated the test run for the original Honeyd 1.5c and compared the results with the IPv6 and IPv4 requests of HoneydV6.

As shown in Table 3, the original Honeyd version and HoneydV6 were able to process about 212 IPv4 requests/s. HoneydV6 managed to handle about 205 IPv6 request/s without any packet loss which is currently more than sufficient in an IPv6 network and

[3]Download avaible from http://www.salbnet.org/

Table 3: Comparison of the number of HTTP GET requests per second that Honeyd 1.5c and HoneydV6 is able to handle without any packet loss.

1.5c (IPv4)	V6 (IPv4)	V6 (IPv6)
212.57	214.00	205.75

only slightly less than its IPv4 counterpart is able to process.

We configured HoneydV6 to simulate just a single target for our test runs. Since Honeyd maintains one connection entry for each connection in a splay tree, regardless of existing connections with the same target address, the performance difference between benchmarking a single target compared to benchmarking multiple targets is insignificant.

8 CONCLUSIONS AND FUTURE WORK

While the general threat level in IPv6 networks is still low compared to IPv4 networks, the results of our IPv6-darknet experiment show the raising interest of attackers in IPv6.

The honeypot *HoneydV6* presented in this paper provides an excellent foundation for future IPv6 network security research. It can be used to observe attacks in IPv6 networks and to reveal new network scan approaches. *HoneydV6* is based on the well-known honeypot *Honeyd* which is the fundamental part of a number of honeypot solutions like *Tiny Honeypot* or the *SCADA HoneyNet Project*. These projects can easily be extended to IPv6 networks using *HoneydV6*.

HoneydV6 is the first low-interaction honeypot which is able to simulate entire IPv6 networks. Besides IPv6 packet processing, *HoneydV6* implements necessary parts of the ICMPv6 and the Neighbor Discovery Protocol. In order to observe new kinds of scanning methods in IPv6 networks, we adapted the internal routing mechanisms of *Honeyd* to support IPv6 packet routing. In our performance tests *HoneydV6* performed comparable to *Honeyd* for both, IPv4 and IPv6 networks. Further, we developed a mechanism that randomly and dynamically generates low-interaction IPv6 hosts, based on the requests of an attacker, in order to increase the chances that an attacker will encounter the honeypot within the huge IPv6 address space.

We are currently setting up a honeynet based on *HoneydV6* together with research partners to observe how the threat level in IPv6 networks develops.

Honeyd still contains some features that are supported in IPv4 networks only. One example is the operating system fingerprinting mechanism, which allows *Honeyd* to emulate system-specific behavior. We currently investigate how the new nmap IPv6 fingerprint format (Nmap, nd) can be reused to simulate the network stack parameters of different operating systems. *HoneydV6* is a useful tool to deceive attackers and to analyse how an attacker interacts with network services. However, the honeypot is not able to inspect UDP or TCP payload for malicious content which makes it hard to extract new exploits from the received traffic. We are therefore working on a connection between our IPv6 honeypot and the shellcode detection library *libemu* (Baecher and Koetter, nd) with the aim of simplifying remote exploit detection.

In order to promote further IPv6 research, we will make the sources of our *HoneydV6* implementation publicly available at http://www.idsv6.de.

REFERENCES

Baecher, P. and Koetter, M. (nd). libemu x86 Shellcode Emulation. Available from: http://libemu.carnivore.it/.

CERT Polska (2012). ENISA Honeypot Study - Proactive Detection of Security Incidents.

Chown, T. (2008). IPv6 Implications for Network Scanning. RFC 5157 (Informational). Available from: http://www.ietf.org/rfc/rfc5157.txt.

Clemente, P., Lalande, J.-F., and Rouzaud-Cornabas, J. (2012). HoneyCloud: Elastic Honeypots - On-attack Provisioning of High-Interaction Honeypots. In *International Conference on Security and Cryptography*, pages 434–439, Rome, Italy.

Dionaea (nd). dionaea catches bugs. Available from: http://dionaea.carnivore.it/.

Ford, M., Stevens, J., and Ronan, J. (2006). Initial Results from an IPv6 Darknet. In *ICISP '06: Proceedings of the International Conference on Internet Surveillance and Protection*, page 13, Washington, DC, USA. IEEE Computer Society.

Heuse, M. (nd). THC IPv6 attack tool kit. Available from: http://www.thc.org/thc-ipv6/ [cited 09.05.2011].

Huston, G. (2010). Background Radiation in IPv6. Available from: https://labs.ripe.net/Members/mirjam/background-radiation-in-ipv6.

Johns, M. S. (1993). Identification Protocol. RFC 1413 (Proposed Standard). Available from: http://www.ietf.org/rfc/rfc1413.txt.

Kalt, C. (2000). Internet Relay Chat: Architecture. RFC 2810 (Informational). Available from: http://www.ietf.org/rfc/rfc2810.txt.

Nmap (nd). Nmap Network Scanning - IPv6 fingerprinting. Available from: http://nmap.org/book/osdetect-ipv6-methods.html.

Oikarinen, J. and Reed, D. (1993). Internet Relay Chat Protocol. RFC 1459. Updated by RFCs 2810, 2811, 2812, 2813. Available from: http://www.ietf.org/rfc/rfc1459.txt.

Pang, R., Yegneswaran, V., Barford, P., Paxson, V., and Peterson, L. (2004). Characteristics of internet background radiation. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, IMC '04, pages 27–40, New York, NY, USA. ACM.

Provos, N. (2003). Honeyd: A Virtual Honeypot Daemon. Technical report, Center for Information Technology Integration, University of Michigan.

Provos, N. and Holz, T. (2008). *Virtual Honeypots - From Botnet Tracking to Intrusion Detection*. Addison-Wesley.

Seifert, C., Welch, I., and Komisarczuk, P. (2006). Taxonomy of honeypots. Technical report, Victoria University of Wellington, Wellington.

SI6 Networks (2012). SI6 Networks' IPv6 Toolkit - A security assessment and troubleshooting tool for the IPv6 protocols. Available from: http://www.si6networks.com/tools/ipv6toolkit.

Sleator, D. D. and Tarjan, R. E. (1985). Self-adjusting binary search trees. *J. ACM*, 32(3):652–686. Available from: http://doi.acm.org/10.1145/3828.3835.

Thomson, S., Narten, T., and Jinmei, T. (2007). IPv6 Stateless Address Autoconfiguration. RFC 4862, Internet Engineering Task Force. Available from: http://tools.ietf.org/html/rfc4862.

Vrable, M., Ma, J., Chen, J., Moore, D., Vandekieft, E., Snoeren, A. C., Voelker, G. M., and Savage, S. (2005). Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 148–162, New York, NY, USA. ACM.

Zinke, J., Habenschuß, J., and Schnor, B. (2012). servload: Generating Representative Workloads for Web Server Benchmarking. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECT)*, Genoa.