

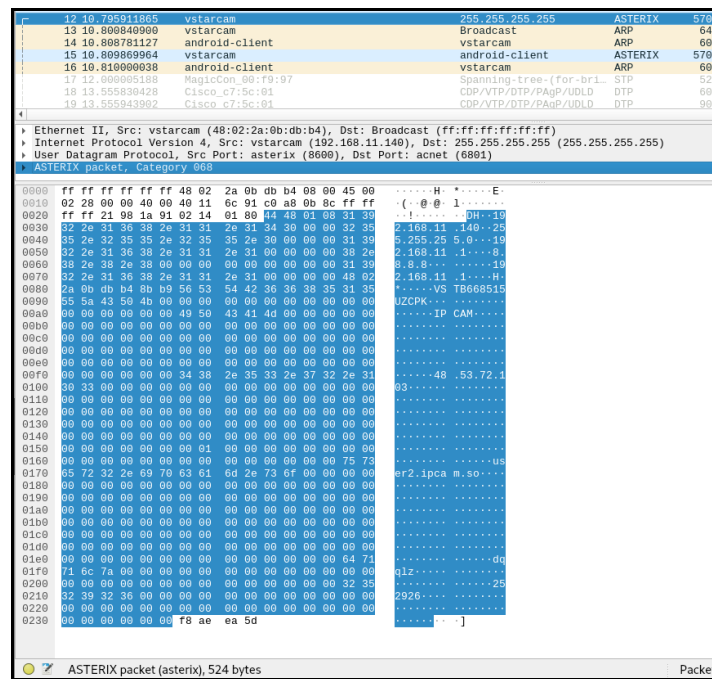
Date: 2019-09-03 Author: Ian Rash Index: #5

We are really hauling through this, now it's time to see if we can replay a GET request via UDP. For this, we will want to go through our captures and grab a couple GET request data packets, including the 16 byte header in the front of the data.

The best way to do this in Wireshark is to find the packets using the filter

* frame contains GET

Then go to the details pane (it's the one above the hex dump and below the packet stream), right click on the Data heading, click Copy, then click As Escaped String. For this I will choose the first UDP GET request in the search.



Once we have the string copied, we can write our simple code to replay the packet.

```
CHECK_USERS_REPLAY = "\xf1\xd0\x00\x68\xd1\x00\x00\x00\x01\x0a\x00\x00\x5c\x00\x00\x00" \
"\x47\x45\x54\x20\x2f\x63\x68\x65\x63\x6b\x5f\x75\x73\x65\x72\x2e" \
"\x63\x67\x69\x3f\x6e\x61\x6d\x65\x3d\x31\x32\x33\x34\x35\x36\x37" \
"\x38\x39\x26\x6c\x6f\x67\x69\x6e\x75\x73\x65\x3d\x61\x64\x6d\x69" \
"\x6e\x26\x6c\x6f\x67\x69\x6e\x70\x61\x73\x3d\x70\x61\x73\x73\x77" \
"\x6f\x72\x64\x26\x75\x73\x65\x72\x3d\x61\x64\x6d\x69\x6e\x26\x70" \
"\x77\x64\x3d\x70\x61\x73\x73\x77\x6f\x72\x64\x26"
def send_replay
  data_sock.send(CHECK_USERS_REPLAY, target)
  LOG.info "Sent replay packet"
end
```

```
require './client'
client = Client.new
client.run

until client.state == :main_phase
  sleep 0.1
end

client.send_replay
sleep 5
client.close
```

After our GET request was sent, we should have gotten two new packets to inspect, some sort of acknowledgment and the results of the command.

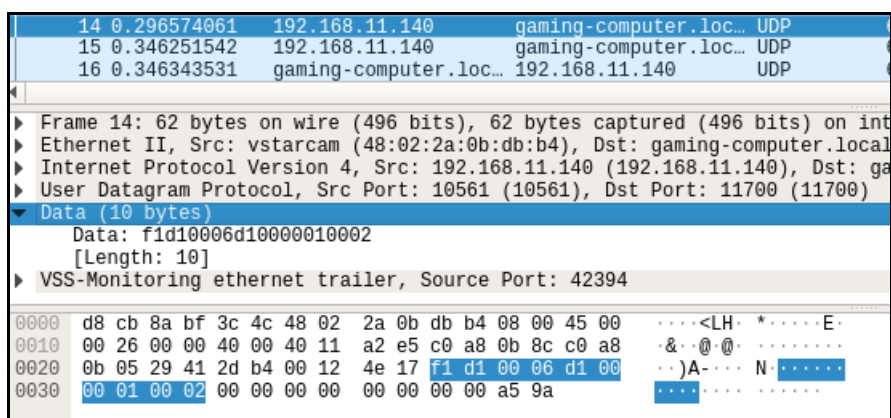
21	3.294631481	vstarcam	gamingcomputer.local	UDP
22	3.834861234	vstarcam	114.55.218.176	UDP
23	3.834975157	vstarcam	47.91.90.222	UDP

▶ Frame 21: 120 bytes on wire (960 bits), 120 bytes captured (960 bits) on				
▶ Ethernet II, Src: vstarcam (48:02:2a:0b:db:b4), Dst: gamingcomputer.local				
▶ Internet Protocol Version 4, Src: vstarcam (192.168.11.140), Dst: gaming				
▶ User Datagram Protocol, Src Port: 10560 (10560), Dst Port: 11700 (11700)				
▼ Data (76 bytes)				
Data: f1d00048d1000000010aa0603c000001726573756c743d20...				
0000	d8 cb 8a bf 3c 4c 48 02	2a 0b db b4 08 00 45 00	<LH*....E.	
0010	00 68 00 00 40 00 40 11	a2 a3 c0 a8 0b 8c c0 a8	.h..@..@.....	
0020	0b 05 29 40 2d b4 00 54	59 32 f1 d0 00 48 d1 00	..)@...T Y2...H..	
0030	00 00 01 0a a0 60 3c 00	00 01 72 65 73 75 6c 74	<..result	
0040	3d 20 30 3b 0d 0a 76 61	72 20 63 75 72 72 65 6e	= 0;..va r curren	
0050	74 5f 75 73 65 72 73 3d	31 3b 0d 0a 76 61 72 20	t_users= 1;..var	
0060	6d 61 78 5f 73 75 70 70	6f 72 74 5f 75 73 65 72	max_supp ort_user	
0070	73 3d 34 3b 0d 0a 24 08		s=4;..\$.	

```
GET_STATUS_REPLAY = "\xf1\xd0\x00\x54\xd1\x00\x00\x02\x01\x0a\x00\x00\x48\x00\x00\x00" \
"\x47\x45\x54\x20\x2f\x67\x65\x74\x5f\x73\x74\x61\x74\x75\x73\x2e" \
"\x63\x67\x69\x3f\x6c\x6f\x67\x69\x6e\x75\x73\x65\x3d\x61\x64\x6d" \
"\x69\x6e\x26\x6c\x6f\x67\x69\x6e\x70\x61\x73\x3d\x38\x38\x38\x38" \
"\x38\x38\x26\x75\x73\x65\x72\x3d\x61\x64\x6d\x69\x6e\x26\x70\x77" \
"\x64\x3d\x38\x38\x38\x38\x38\x38"
```

```
def send_replay
    data_sock.send(GET_STATUS_REPLAY, target)
    LOG.info "Sent replay packet"
end
```

This time when we send the replay let's see what happens.



This time we get something a little different. We got the "acknowledgement" packet but we didn't get any results. Upon closer inspect we can see the two acknowledgement packets are just slightly different, the last byte on our success being 0x00 and the last byte on our failure was 0x02. We now know that the order of the packets is important. This could signify that the mysterious header has some values in it that track order of packets.

Let's learn more.

The next thing we are going to want to try is to modify a request, and see if we can get it to teach us something new about the protocol.

```
CHECK_USERS_HEADER = "\xf1\xd0\x00\x68\xd1\x00\x00\x00\x01\x0a\x00\x00\x5c\x00\x00\x00"
CHECK_USERS_REQUEST = "GET /check_user.cgi?name=123456789&loginuse=admin&loginpas=password&user=admin"
CHECK_USERS_MODIFIED_REQUEST1 = "GET /check_user.cgi?name=44444&loginuse=admin&loginpas=password&user=admin"
CHECK_USERS_MODIFIED_REQUEST2 = "GET /check_user.cgi?name=1234567890&loginuse=admin&loginpas=password&user=admin"
CHECK_USERS_MODIFIED_REQUEST3 = "GET /check_user.cgi?name=987654321&loginuse=admin&loginpas=password&user=admin"
CHECK_USERS_REPLAY = CHECK_USERS_HEADER + CHECK_USERS_REQUEST
CHECK_USERS_MODIFIED_REPLAY1 = CHECK_USERS_HEADER + CHECK_USERS_MODIFIED_REQUEST1
CHECK_USERS_MODIFIED_REPLAY2 = CHECK_USERS_HEADER + CHECK_USERS_MODIFIED_REQUEST2
CHECK_USERS_MODIFIED_REPLAY3 = CHECK_USERS_HEADER + CHECK_USERS_MODIFIED_REQUEST3
```

In the constant CHECK_USERS_MODIFIED_REQUEST1 I changed the name parameter to make it shorter, and in CHECK_USERS_MODIFIED_REQUEST2 I made the name a bit longer, and in CHECK_USERS_MODIFIED_REQUEST3 I reversed the name, but kept the amount of chars the same.

Sending 1 or 2 does nothing and the server doesn't even reply back. Sending request 3 however, works just fine, and produces the correct output.

What we just learned from this is that the header has values specifically related to size not content.

If we try to replay the same packet twice in a session, we see another weird behavior.

```
I, [2019-03-05 08:37:30 -08:00 #18513] INFO -- : Sent replay packet
I, [2019-03-05 08:37:30 -08:00 #18513] INFO -- : Sent Pong
I, [2019-03-05 08:37:30 -08:00 #18513] INFO -- : UNKNOWN PACKET RECEIVED from 192.168.11.140:10560 : f1\xd1\x00\x
I, [2019-03-05 08:37:30 -08:00 #18513] INFO -- : UNKNOWN PACKET RECEIVED from 192.168.11.140:10560 : f1\xd0\x00\x
I, [2019-03-05 08:37:30 -08:00 #18513] INFO -- : Sent Pong
I, [2019-03-05 08:37:31 -08:00 #18513] INFO -- : Sent Pong
I, [2019-03-05 08:37:32 -08:00 #18513] INFO -- : Sent Pong
I, [2019-03-05 08:37:32 -08:00 #18513] INFO -- : Sent Pong
I, [2019-03-05 08:37:34 -08:00 #18513] INFO -- : Sent Pong
I, [2019-03-05 08:37:34 -08:00 #18513] INFO -- : Sent replay packet
I, [2019-03-05 08:37:34 -08:00 #18513] INFO -- : UNKNOWN PACKET RECEIVED from 192.168.11.140:10560 : f1\xd1\x00\x
I, [2019-03-05 08:37:34 -08:00 #18513] INFO -- : Sent Pong
I, [2019-03-05 08:37:34 -08:00 #18513] INFO -- : Sent Pong
```

The first replay works fine, but the second one doesn't produce results, but does produce the failure acknowledgement. Maybe the acknowledgment packet signifies that the is formatted correctly, but won't give the results if the values aren't 100% correct. My guess would be that the bytes that control size are fine, but the bytes that control order are not.

Let's move on to deciphering these two mysteries.

Figuring out the header

The first big mystery we want to solve is the header, how it's made, and what we can do with it. To do this, we will open up Android Studio with the Eye4 app and do a full capture, from logging into the device, to changing all the settings in the client.

Here are some examples, I pulled them all in order from when they were received. I also do a little number analysis on it and print out the result.

```
CHECK_USERS_HEADER = "\xf1\xd0\x00\x68\xd1\x00\x00\x00"
CHECK_USERS_REQUEST_HEADER = "\x01\x0a\x00\x00\x5c\x00\x00\x00"
CHECK_USERS_REQUEST = "GET /check_user.cgi?name=123456789&loginuse=admin&loginpas=password&user=admin"
CHECK_USERS_REPLAY = CHECK_USERS_HEADER + CHECK_USERS_REQUEST_HEADER + CHECK_USERS_REQUEST

CONGLOMERATE_HEADER = "\xf1\xd0\x01\xb9\xd1\x00\x00\x01"
CONGLOMERATE_REQUEST1_HEADER = "\x01\x0a\x00\x00\x51\x00\x00\x00"
```

```
CONGLOMERATE_REQUEST1 = "GET /snapshot.cgi?res=1&loginuse=admin&loginpas=password&user=admin&pwd=p
CONGLOMERATE_REQUEST2_HEADER = "\x01\x0a\x00\x00\x4c\x00\x00\x00"
CONGLOMERATE_REQUEST2 = "GET /get_status.cgi?loginuse=admin&loginpas=password&user=admin&pwd=passw
CONGLOMERATE_REQUEST3_HEADER = "\x01\x0a\x00\x00\x53\x00\x00\x00"
CONGLOMERATE_REQUEST3 = "GET /get_factory_param.cgi?loginuse=admin&loginpas=password&user=admin&pv
CONGLOMERATE_REQUEST4_HEADER = "\x01\x0a\x00\x00\x4c\x00\x00\x00"
CONGLOMERATE_REQUEST4 = "GET /get_params.cgi?loginuse=admin&loginpas=password&user=admin&pwd=pass
CONGLOMERATE_REQUEST5_HEADER = "\x01\x0a\x00\x00\x51\x00\x00\x00"
CONGLOMERATE_REQUEST5 = "GET /snapshot.cgi?&res=1&loginuse=admin&loginpas=password&user=admin&pwd=
CONGLOMERATE_REPLAY = CONGLOMERATE_HEADER + CONGLOMERATE_REQUEST1_HEADER + CONGLOMERATE
CONGLOMERATE_REQUEST3_HEADER + CONGLOMERATE_REQUEST3 + CONGLOMERATE_REQUEST4_
```

```
SET_FACTORY_HEADER = "\xf1\xd0\x00\x7e\xd1\x00\x00\x02"
SET_FACTORY_REQUEST_HEADER = "\x01\x0a\x00\x00\x72\x00\x00\x00"
SET_FACTORY_REQUEST = "GET /set_factory_param.cgi?alarm_server=push.eye4.cn/VSTC&loginuse=admin&login
SET_FACTORY_REPLAY = SET_FACTORY_HEADER + SET_FACTORY_REQUEST_HEADER + SET_FACTORY_REQUEST
```

```
SET_DATETIME_HEADER = "\xf1\xd0\x00\x99\xd1\x00\x00\x03"
SET_DATETIME_REQUEST_HEADER = "\x01\x0a\x00\x00\x8d\x00\x00\x00"
SET_DATETIME_REQUEST = "GET /set_datetime.cgi?tz=28800&ntp_enable=1&ntp_svr=time.windows.com&now=155
SET_DATETIME_REPLAY = SET_DATETIME_HEADER + SET_DATETIME_REQUEST_HEADER + SET_DATETIME_REQUEST
```

```
puts "CHECK_USERS_REPLAY BREAKDOWN"
puts "  REQUEST LENGTH = D:#{CHECK_USERS_REQUEST.size} | H:0x#{CHECK_USERS_REQUEST.size.to_s(16)}"
puts "  PACKET LENGTH = D:#{CHECK_USERS_REPLAY.size} | H:0x#{CHECK_USERS_REPLAY.size.to_s(16)}"
puts
puts "CONGLOMERATE_REPLAY BREAKDOWN"
puts "  REQUEST1 LENGTH = D:#{CONGLOMERATE_REQUEST1.size} | H:0x#{CONGLOMERATE_REQUEST1.size.to_s(16)}"
puts "  REQUEST2 LENGTH = D:#{CONGLOMERATE_REQUEST2.size} | H:0x#{CONGLOMERATE_REQUEST2.size.to_s(16)}"
puts "  REQUEST3 LENGTH = D:#{CONGLOMERATE_REQUEST3.size} | H:0x#{CONGLOMERATE_REQUEST3.size.to_s(16)}"
puts "  REQUEST4 LENGTH = D:#{CONGLOMERATE_REQUEST4.size} | H:0x#{CONGLOMERATE_REQUEST4.size.to_s(16)}"
puts "  REQUEST5 LENGTH = D:#{CONGLOMERATE_REQUEST5.size} | H:0x#{CONGLOMERATE_REQUEST5.size.to_s(16)}"
puts "  PACKET LENGTH = D:#{CONGLOMERATE_REPLAY.size} | H:0x#{CONGLOMERATE_REPLAY.size.to_s(16)}"
puts
puts "SET_FACTORY_REPLAY BREAKDOWN"
puts "  REQUEST LENGTH = D:#{SET_FACTORY_REQUEST.size} | H:0x#{SET_FACTORY_REQUEST.size.to_s(16)}"
puts "  PACKET LENGTH = D:#{SET_FACTORY_REPLAY.size} | H:0x#{SET_FACTORY_REPLAY.size.to_s(16)}"
puts
puts "SET_DATETIME_REPLAY BREAKDOWN"
puts "  REQUEST LENGTH = D:#{SET_DATETIME_REQUEST.size} | H:0x#{SET_DATETIME_REQUEST.size.to_s(16)}"
puts "  PACKET LENGTH = D:#{SET_DATETIME_REPLAY.size} | H:0x#{SET_DATETIME_REPLAY.size.to_s(16)}"
```

CHECK_USERS_REPLAY BREAKDOWN

REQUEST LENGTH = D:92 | H:0x5c

PACKET LENGTH = D:108 | H:0x6c

CONGLOMERATE_REPLAY BREAKDOWN

REQUEST1 LENGTH = D:81 | H:0x51

REQUEST2 LENGTH = D:76 | H:0x4c

REQUEST3 LENGTH = D:83 | H:0x53

REQUEST4 LENGTH = D:76 | H:0x4c

REQUEST5 LENGTH = D:81 | H:0x51

PACKET LENGTH = D:445 | H:0x1bd

SET_FACTORY_REPLAY BREAKDOWN

REQUEST LENGTH = D:114 | H:0x72

PACKET LENGTH = D:130 | H:0x82

SET_DATETIME_REPLAY BREAKDOWN

REQUEST LENGTH = D:141 | H:0x8d

PACKET LENGTH = D:157 | H:0x9d

[Done] exited with code=0 in 0.663 seconds

When writing this test, I noticed that the 8 byte segments were very similar to the other 8 byte segments, so I wanted to break them up, because I was sure it was significant.

We want to take a look at the hex numbers and start comparing them to numbers in the header. We see, for example, that CHECK_USER_REQUEST length is 0x5c, which we can also see in the CHECK_USER_REQUEST_HEADER in byte[4]. We can see this in every request header.

Another interesting thing we can notice is that the CHECK_USERS_HEADER has a byte, very close to 0x5c, in fact, only 0x4 off. If we go through the other packets (except for the conglomerate one), we see this is the case every time.

Taking a look at the conglomerate packet confirms our suspicion that the 8 bytes right before the GET request is tied to the request itself. We can also see that the top 8 byte header contains 0x1b9 which is 0x4 off the total length 0x1bd. We can also see the request headers match up with the total bytes in each separate request. We also can see the top header's byte length is a 2 byte big endian integer, so we will need to plan for that.

Each of those packets were taken sequentially, in order, from the capture. We can see the last most byte in each of the headers denotes what packet order it's on. We will need to keep track of the number of requests we send.

Ultimately, this all means that we can forge requests, all we need is the GET request length, and the number of GET requests the client has sent.

```
USER = "admin"
PASS = "password"
LOGIN_PARAMS = "&loginuse=#{USER}&loginpas=#{PASS}&user=#{USER}&pwd=#{PASS}"
@requests_sent = 0

def make_udp_header(get_request)
  "\xf1\xd0#{String.new(Bytes[get_request.size + 0xc]).rjust(2, "\x00"[0])}\xd1\x00#{String.new(Bytes[@requests_s
end

def make_get_request_header(get_request)
  "\x01\x0a\x00#{String.new(Bytes[get_request.size]).rjust(2, "\x00"[0])}\x00\x00\x00"
end

def send_udp_get_request(cgi : String, **params)
  param_string = params.keys.map{|param_name| "#{param_name}=#{params[param_name]}"}.join('&')
  get_request = "GET /#{cgi}.cgi?#{param_string}#{LOGIN_PARAMS}"
  header = make_udp_header(get_request)
  request_header = make_get_request_header(get_request)
  data_sock.send(header + request_header + get_request, target)
  @requests_sent += 1
  LOG.info "SENT #{get_request}"
end
```

Forging Requests

Forging our own packets now seems pretty plausible, let's give it a try.

```
require "./client"
```

```
client = Client.new
```

```
client.run
```

```
# Wait until main phase
```

```
until client.state == :main_phase
```

```
  sleep 0.1
```

```
end
```

```
client.send_udp_get_request("check_user", name: "123456789")
```

```
sleep 3
```

```
client.send_udp_get_request("check_user", name: "123456789")
```

```
sleep 3
```

```
client.send_udp_get_request("check_user", name: "123456789")
```

```
sleep 3
```

```
client.send_udp_get_request("check_user", name: "123456789")
```

```
sleep 3
```

```
sleep 5
```

```
client.close
```

This code will not only test if we can forge the first packet, but also the subsequent ones. Checking the Wireshark with a special filter shows the success * frame contains GET || frame contains F1:D1:00 || frame contains F1:D0:00

No.	Time	Source	Destination	Protocol	Length	Info
11	0.104989908	gamingcomputer.local	vstarcam	UDP	151	11700 → 10561 Len=107
14	0.292015238	vstarcam	gamingcomputer.local	UDP	62	10561 → 11700 Len=10
17	0.394214381	vstarcam	gamingcomputer.local	UDP	120	10561 → 11700 Len=76
39	3.108454094	gamingcomputer.local	vstarcam	UDP	151	11700 → 10561 Len=107
40	3.121364177	vstarcam	gamingcomputer.local	UDP	62	10561 → 11700 Len=10
43	3.221366842	vstarcam	gamingcomputer.local	UDP	120	10561 → 11700 Len=76
60	6.112250157	gamingcomputer.local	vstarcam	UDP	151	11700 → 10561 Len=107
61	6.121454302	vstarcam	gamingcomputer.local	UDP	62	10561 → 11700 Len=10
64	6.201893326	vstarcam	gamingcomputer.local	UDP	120	10561 → 11700 Len=76
73	9.116646775	gamingcomputer.local	vstarcam	UDP	151	11700 → 10561 Len=107
74	9.124132213	vstarcam	gamingcomputer.local	UDP	62	10561 → 11700 Len=10
77	9.201774922	vstarcam	gamingcomputer.local	UDP	120	10561 → 11700 Len=76

▶	Frame 17: 120 bytes on wire (960 bits), 120 bytes captured (960 bits) on interface 0
▶	Ethernet II, Src: vstarcam (48:02:2a:0b:db:b4), Dst: gamingcomputer.local (d8:cb:8a:bf:3c:4c)
▶	Internet Protocol Version 4, Src: vstarcam (192.168.11.140), Dst: gamingcomputer.local (192.168.11.5)
▶	User Datagram Protocol, Src Port: 10561 (10561), Dst Port: 11700 (11700)
▼	Data (76 bytes)
	Data: f1d00048d1000000010aa0603c000001726573756c743d20...
	[Length: 76]
▶	VSS-Monitoring ethernet trailer, Source Port: 9224

0000	d8 cb 8a bf 3c 4c 48 02	2a 0b db b4 08 00 45 00	<LH. *....E.
0010	00 68 00 00 40 00 40 11	a2 a3 c0 a8 0b 8c c0 a8	..h..@..@.....
0020	0b 05 29 41 2d b4 00 54	59 31 f1 d0 00 48 d1 00	..)A-..T Y1...H..
0030	00 00 01 0a a0 60 3c 00	00 01 72 65 73 75 6c 74	<..result
0040	3d 20 30 3b 0d 0a 76 61	72 20 63 75 72 72 65 6e	= 0;..va r curren
0050	74 5f 75 73 65 72 73 3d	31 3b 0d 0a 76 61 72 20	t_users= 1;..var
0060	6d 61 78 5f 73 75 70 70	6f 72 74 5f 75 73 65 72	max_supp ort_user
0070	73 3d 34 3b 0d 0a 24 08		s=4;..\$.

TAGGED SKILLS:

Reverse Engineering

Static Analysis

C

