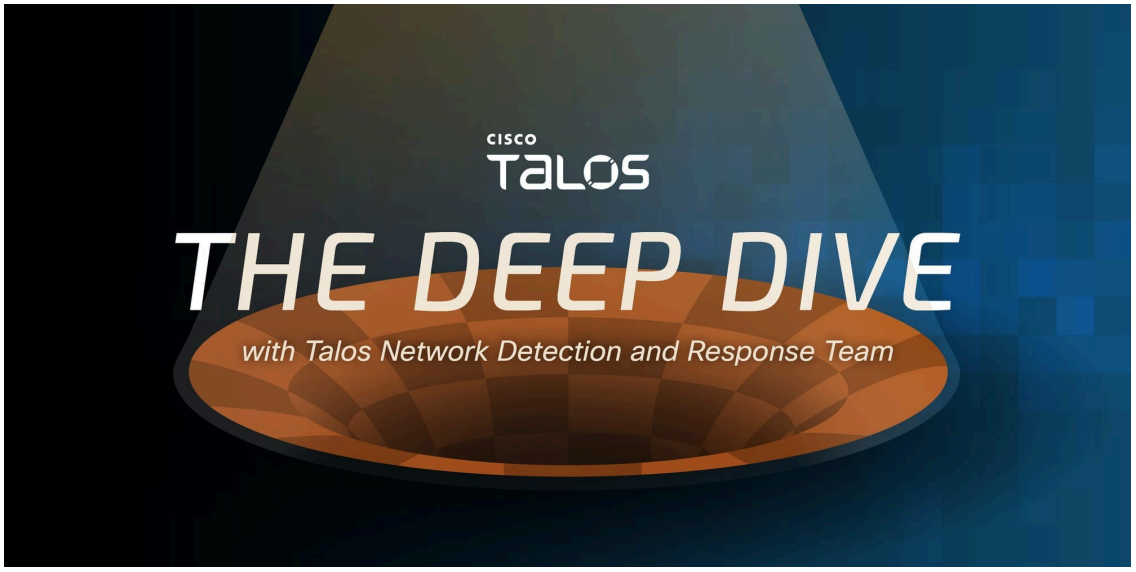
Writing a BugSleep C2 server and detecting its traffic with Snort

By Aaron Boyd

Published: 2024-10-30 · Archived: 2026-04-05 19:16:45 UTC



Wednesday, October 30, 2024 06:00

In June 2024, security researchers published their analysis of a novel implant dubbed “MuddyRot”(aka "BugSleep"). This remote access tool (RAT) gives operators reverse shell and file input/output (I/O) capabilities on a victim’s endpoint using a bespoke command and control (C2) protocol. This blog will demonstrate the practice and methodology of reversing BugSleep’s protocol, writing a functional C2 server, and detecting this traffic with Snort.

Key findings

- BugSleep implant implements a bespoke C2 protocol over plain TCP sockets.
- BugSleep operators have demonstrated multiple file-obfuscation techniques to avoid detection.
- BugSleep implements reverse shell, file I/O, and persistence capabilities on the target system.

Sending and receiving data

This blog will use sample b8703744744555ad841f922995cef5dbca11da22565195d05529f5f9095fbfca for analysis. Two of the lowest functions in the C2 stack, referred to as SendSocket (FUN_1400034c0) and ReadSocket (FUN_140003390), are very light wrappers for the send and receive API functions and handle payload encryption. They include some error handling by attempting to send or receive data 10 times before failing.

This protocol uses a pseudo-TLV (Type Length Value) structure with only two types: integer or string. Integers are sent as little-endian 4- or 8-byte values, and strings are prepended with the 4-byte value of its length. Payloads are then encrypted by subtracting a static value from each byte in the buffer (in this sample it is three).

Type	Value	Plain text	Cipher text
IntegerMsg	6	06 00 00 00	03 FD FD FD
StringMsg	Talos	05 00 00 00 48 65 6C 6C 6F	02 FD FD FD 51 5E 69 6C 70

Figure 1: Example of data encryption used by BugSleep

There are two main functions for handling C2 communications: C2Loop (FUN_1400012c0) and CommandHandler (FUN_1400028a0). C2Loop is responsible for setting up socket connections with the server and sending a beacon, while CommandHandler is responsible for processing and executing commands from the server.

After setting up the socket connection, the implant beacons (FUN_140003d80) to the C2 server for a command. The beacon is a StringMsg in the form ComputerName/Username. If the server responds with an IntegerMsg equal to 0x03, BugSleep will terminate itself. We suspect this is remnants of an old kill command or an emergency kill without the overhead of reading the real kill command later.

Each BugSleep command is sent as an IntegerMsg after the beacon response. The following enumeration defines all the command IDs discovered.

```
class BugSleepCmd(IntEnum):
    GET              = 0x01  # (path): download file from system
    PUT              = 0x02  # (path, data): upload file to system
    SHELL            = 0x03  # (void): create reverse shell session
    INC_TIMEOUT      = 0x04  # (int): add X seconds to socket timeout
    KILL             = 0x06  # (void): terminate BugSleep process
    TASK_DELETE      = 0x09  # (void): delete BugSleep scheduled task
    TASK_CHECK       = 0x0A  # (void): is BugSleep task installed
    TASK_ADD         = 0x0B  # (void): install BugSleep scheduled task
    UPDATE_HEARTBEAT = 0x61  # (int): seconds between beacons
    SOCK_TIMEOUT     = 0x62  # (int): C2 socket timeout in seconds
    PING             = 0x63  # (void): ping BugSleep instance
```

Figure 2: Command IDs used by implant

Phoning home

The implant communicates using plain TCP sockets, which can be seen using a Netcat listener and Wireshark.

```
Wireshark · Follow TCP Stream (tcp.stream eq 1) · any

00000000  11 fd fd fd                                        ....
00000004  41 42 50 48 51 4c 4d 2a  53 35 2f 45 2d 52 52 2c  ABPHQLM* S5/E-RR,
00000014  52 70 62 6f                                       Rpbo
```

Figure 3: BugSleep beacon as seen through Wireshark.

Recalling the message encryption demonstrated in Figure 1, the beacon can be decrypted with a little bit of Python (Figure 4). This will be used again when building the rest of the C2 server.

```
In [1]: import struct

In [2]: def process_payload(payload, offset=3):
   ...:     """Decode data sent from BugSleep."""
   ...:     r = b""
   ...:     for c in payload:
   ...:         r += int.to_bytes((c + offset) & 0xFF, 1, "little")
   ...:     return r
   ...:

In [2]: struct.unpack("<I", process_payload(b"\x11\xfd\xfd\xfd", 3))
Out[2]: (20,)

In [3]: process_payload(b"ABPHQLM*S5/E-RR,Rpbo", 3)
Out[3]: b'DESKTOP-V82H0UU/User'

In [4]: len(process_payload(b"ABPHQLM*S5/E-RR,Rpbo", 3))
Out[4]: 20
```

Figure 4: Decoding beacon data

Python C2 server

With an understanding of the protocol basics, it is time to start building the C2 server. Full source code can be found here.

Beacon

As mentioned earlier, the BugSleep beacon function sends a StringMsg and reads an IntegerMsg response from the server. Since the IntegerMsg returned can be anything but 0x03, we returned the length of the Computer Name/Username string received by the server.

```
user@ubuntu:~$ sudo ./c2-server/server.py -v
[2024-08-05 11:06:45,014][BugSleep][INFO    ] Listening on 0.0.0.0:443...
[2024-08-05 11:06:45,682][BugSleep][INFO    ] Connected by 172.10.10.3:29417
[2024-08-05 11:06:45,682][BugSleep][INFO    ] Reading beacon data...
[2024-08-05 11:06:45,729][BugSleep][DEBUG   ] Received:
[2024-08-05 11:06:45,729][BugSleep][DEBUG   ]   plain(14 00 00 00)
[2024-08-05 11:06:45,729][BugSleep][DEBUG   ]   cipher(11 FD FD FD)
[2024-08-05 11:06:45,729][BugSleep][DEBUG   ] Received:
[2024-08-05 11:06:45,729][BugSleep][DEBUG   ]   plain(DESKTOP-V82H0UU/User)
[2024-08-05 11:06:45,729][BugSleep][DEBUG   ]   cipher(ABPHQLM*S5/E-RR,Rpbo)
[2024-08-05 11:06:45,729][BugSleep][DEBUG   ] Sent:
[2024-08-05 11:06:45,729][BugSleep][DEBUG   ]   plain(14 00 00 00)
[2024-08-05 11:06:45,729][BugSleep][DEBUG   ]   cipher(17 03 03 03)
```

Figure 5: Output from C2 server receiving beacon data

Ping command

The simplest command to implement is the Ping command. It has the command ID of 0x63 (BugSleep subtracts one from whatever ID it receives). The code is simple: send back 4 bytes.

```
case 0x62:
    SendSocket(ret,(char *)lpBuffer,4);
```

Figure 6: Switch case for handling ping command

Once the beacon comes in, the server is responsible for:

1. Sending 4 bytes for beacon response
2. Sending 4 bytes for Ping command ID
3. Reading 4 bytes of Ping data

The ping command was observed sending back 4 bytes recently allocated on the heap, so it's not guaranteed to know what that data looks like. To validate things are really working, a breakpoint can be set in WinDbg and memory set manually before being sent.

```
user@ubuntu:~$ sudo ./c2-server/server.py -v
[2024-08-05 11:09:45,568][BugSleep][INFO    ] Listening on 0.0.0.0:443...
[2024-08-05 11:09:45,867][BugSleep][INFO    ] Connected by 172.10.10.3:29419
[2024-08-05 11:09:45,867][BugSleep][INFO    ] Reading beacon data...
[2024-08-05 11:09:45,897][BugSleep][DEBUG   ] Received:
[2024-08-05 11:09:45,898][BugSleep][DEBUG   ]   plain(14 00 00 00)
[2024-08-05 11:09:45,898][BugSleep][DEBUG   ]   cipher(11 FD FD FD)
[2024-08-05 11:09:45,898][BugSleep][DEBUG   ] Received:
[2024-08-05 11:09:45,898][BugSleep][DEBUG   ]   plain(DESKTOP-V82H0UU/User)
[2024-08-05 11:09:45,898][BugSleep][DEBUG   ]   cipher(ABPHQLM*S5/E-RR,Rpbo)
[2024-08-05 11:09:45,898][BugSleep][DEBUG   ] Sent:
[2024-08-05 11:09:45,898][BugSleep][DEBUG   ]   plain(14 00 00 00)
[2024-08-05 11:09:45,898][BugSleep][DEBUG   ]   cipher(17 03 03 03)
[2024-08-05 11:09:45,898][BugSleep][INFO    ] Sending PING...
[2024-08-05 11:09:45,898][BugSleep][DEBUG   ] Sent:
[2024-08-05 11:09:45,898][BugSleep][DEBUG   ]   plain(63 00 00 00)
[2024-08-05 11:09:45,898][BugSleep][DEBUG   ]   cipher(66 03 03 03)
[2024-08-05 11:10:32,848][BugSleep][DEBUG   ] Received:
[2024-08-05 11:10:32,848][BugSleep][DEBUG   ]   plain(DE AD BE EF)
[2024-08-05 11:10:32,848][BugSleep][DEBUG   ]   cipher(DB AA BB EC)
[2024-08-05 11:10:32,848][BugSleep][INFO    ] Received ping response: DE AD BE EF
```

Figure 7: Confirming 0xdeadbeef written to memory is received by the server in a Ping command

File commands

The next set of commands are responsible for downloading files onto the compromised system or uploading files to the C2 server (PutFile and GetFile, respectively). These commands are inverses of each other, so only the GetFile command will be discussed in detail. The methodology was to trace each call to SendSocket or ReadSocket and implement the response for that call in Python. In CommandHandler, the implant reads the length and value off the wire. This is the file to be retrieved.

```
case 0:
    dwRet = ReadSocket(ret,(char *)lpBuffer,4);
    if ((((int)dwRet != -1) &&
        (dwRet = ReadSocket((int)gSocket,(char *)lpC2Buffer,*lpBuffer), (int)dwRet != -1)) &&
        (ret = memcmp(lpC2Buffer,lpExitStr,5), ret != 0)) {
        CmdGetFile((LPCWSTR)lpC2Buffer);
    }
    break;
```

Figure 8: GetFile reading path string length and path string from socket

The CmdGetFile function opens the target file and chunks it over the socket one page at a time. The list of SendSocket calls is as follows:

```
Send(4-byte "success" value)
Send(4-byte zero value (make sure socket is still alive after sleep?))
Send(8-byte number of pages)
Send(4-byte number of bytes used on last page (if len(file) % 0x400 != 0))

For N pages:
   Send(0x400 bytes of file data)

if len(file) % 0x400 != 0:
    Send(remaining file data)
```

Figure 9: SendSocket calls made by CmdGetFile function

```
user@ubuntu:~$ sudo ./c2-server/server.py
[2024-08-05 11:12:02,894][BugSleep][INFO    ] Listening on 0.0.0.0:443...
[2024-08-05 11:12:03,103][BugSleep][INFO    ] Connected by 172.10.10.3:29421
[2024-08-05 11:12:03,103][BugSleep][INFO    ] Reading beacon data...
[2024-08-05 11:12:03,134][BugSleep][INFO    ] Sending GET...
[2024-08-05 11:12:03,259][BugSleep][INFO    ] File is 1 page(s)
[2024-08-05 11:12:03,259][BugSleep][INFO    ] Last page is 760 bytes
[2024-08-05 11:12:03,260][BugSleep][INFO    ] Received 760 bytes.
[2024-08-05 11:12:03,267][BugSleep][INFO    ] Saved file ./target_files/C/Users/User/Documents/Hello.txt
[2024-08-05 11:12:03,337][BugSleep][INFO    ] Connected by 172.10.10.3:29422
[2024-08-05 11:12:03,337][BugSleep][INFO    ] Reading beacon data...
[2024-08-05 11:12:03,369][BugSleep][INFO    ] Sending GET...
[2024-08-05 11:12:03,488][BugSleep][INFO    ] File is 196 page(s)
[2024-08-05 11:12:03,493][BugSleep][INFO    ] Last page is 1024 bytes
[2024-08-05 11:12:03,547][BugSleep][INFO    ] Received 200704 bytes.
[2024-08-05 11:12:03,548][BugSleep][INFO    ] Saved file ./target_files/C/Windows/System32/notepad.exe
user@ubuntu:~$ cat ./target_files/C/Users/User/Documents/Hello.txt
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed eget interdum lacus. Donec tortor
lacus, viverra non tincidunt eget, venenatis in ex. Duis tempor finibus erat at cursus. Proin
consequat cursus tincidunt. Nulla sodales viverra malesuada. Ut cursus et sem at pulvinar.
Phasellus at iaculis dui. Nulla interdum nisi vitae efficitur tempor. Vestibulum eu mi non diam
accumsan mattis. Quisque aliquam lectus tincidunt ex rutrum blandit. Mauris gravida lorem massa,
eu vulputate metus efficitur quis. Cras turpis sem, condimentum vel faucibus id, tincidunt vitae
dui. Nulla ac dignissim odio. Vivamus augue ante, laoreet non molestie lacinia, ultricies vel
erat. Nulla laoreet efficitur tristique. Donec at orci a nunc auctor semper non a nisl.
user@ubuntu:~$ file ./target_files/C/Windows/System32/notepad.exe
./target_files/C/Windows/System32/notepad.exe: PE32+ executable (GUI) x86-64, for MS Windows
user@ubuntu:~$
```

Figure 10: Example C2 server output from GetFile command

The PutFile command differs slightly from the GetFile command with how it uses pointer math to process incoming pages.

```
SetFilePointer(hFile,*tmpHeapBuff * 0x3fc,NULL,0);
TryWriteFile(hFile,(longlong)(tmpHeapBuff + 1),0x3fc,0,NULL);
```

Figure 11: Tricky file pointer math

This translates to each page starting with a 4-byte page number followed by 1020 bytes (or 0x3fc) of file data, which the GetFile command does not do; it sends full 1024-byte pages of file data without page numbers.

Reverse shell

The last command is the reverse shell. This is the most complex because it requires many reads and writes over the socket. The disassembly is rather long and difficult to keep track of the socket calls, so we have omitted it. Effectively, the implant spawns a cmd.exe process (FUN_1400016e0) and reads the command to execute from the socket. The shell command and its output are marshaled between the processes via pipes during the session. The complexity of this operation comes from BugSleep incrementally reporting return values from pipe API calls while attempting to read shell output (FUN_140003840). The implant will enter this loop of reading commands and sending output until it receives the string “terminate\n”.

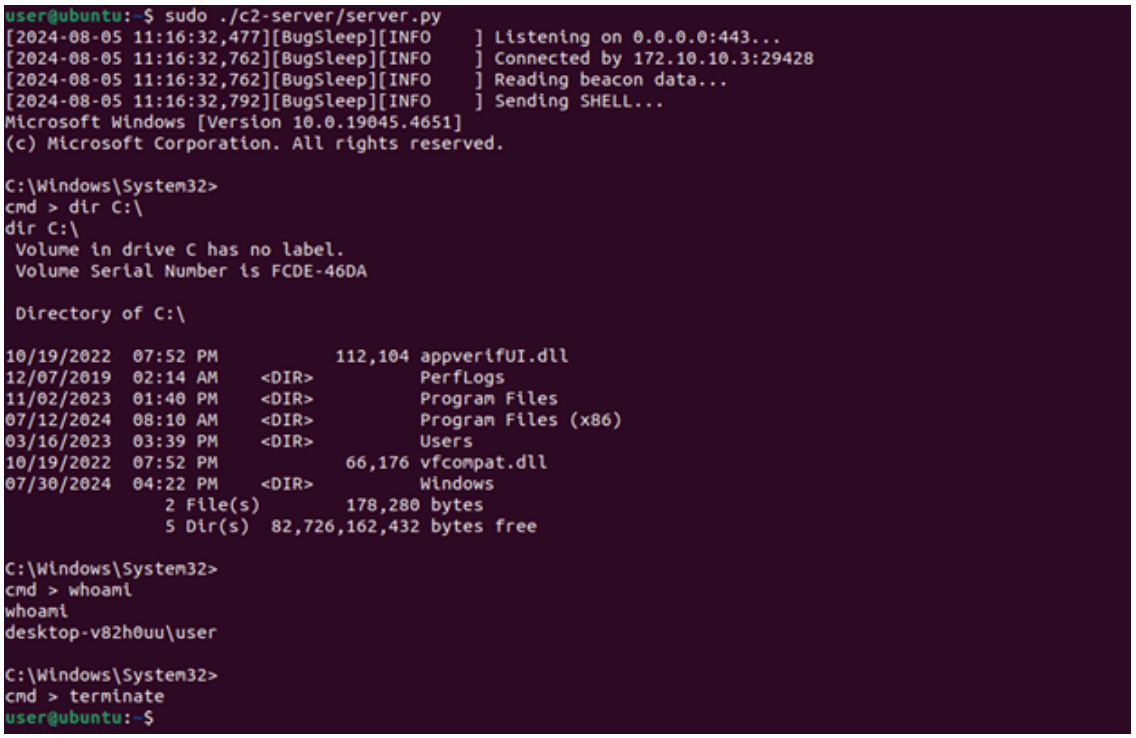


Figure 12: Example output from C2 server running the reverse shell command

The rest of the commands are less complex but have been implemented and are viewable here.

Snort detection

This server gives Talos the ability to emulate any number of conversations between BugSleep and its operators. This traffic is crucial for writing and validating our detections’ performance in the wild.

The initial candidate for detection would be the beacon. It is the first opportunity to shut down communications, isolating any BugSleep instance from receiving commands. It was observed that each beacon has the form of <len><data>, where data is sub_string(COMPUTER_NAME + "/" + USERNAME, 3). This string is not long or static, which makes it a poor candidate for a fast_pattern; however, recall that each beacon is prepended with a 4-byte length of this string. A Computer Name/Username string from any given victim is unlikely to be longer than 255 characters. This means most length fields are going to look like |XX 00 00 00| or |XX FD FD FD| when encoded. This could be a quick match, early in the stream, at a static offset, making it a decent fast_pattern candidate.

```
content:"|FD FD FD|",offset 1,depth 3,fast_pattern;
```

Figure 13: Detecting higher order encoded zero bytes of beacons sent from BugSleep

This will work but is likely to cause false-positives (FP) in the wild. Every sample of BugSleep was seen using port 443. The implant is also reaching outside the network to a C2 server, so traffic to be inspected by this rule can be reduced using the following header:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET 443 ()
```

Figure 14: Restricting rule to inspect traffic leaving the network to port 443

The flow:to_server,established option can be used to restrict Snort to data coming from a client over established TCP streams. The FP-rate on this rule still isn't great. Any TCP traffic leaving the network on port 443 with |FD FD FD| at offset 1 will alert. That might sound unique, but it does not indicate with confidence that the traffic is a BugSleep beacon.

One powerful tool in Snort to add more logic or state to rules is flowbits. These allow a writer to have a sense of state within a stream across multiple rules. In this case, the beacons aren't enough to reliably alert on. What if we use flowbits to chain beacons with the commands being sent back? The commands themselves don't provide much content, as they are variable length non-deterministic strings (e.g., get, put, etc.) or a nondeterministic 4-byte integer (e.g., heartbeat, increment timeout, etc.). They do, however, all start with a 4-byte command ID. Setting a flowbit when a beacon leaves the network will allow another rule down the line to alert with higher confidence if it sees a command ID come back in the same stream.

Command rules

The pcre rule option can be used to reduce 11 rules down to one. Like the beacon rule, the three zero bytes, encoded as |03|, can be used as a fast_pattern. Once the rule has entered, the bugsleep_beacon flowbit check can be performed to help the rule exit quickly in the event of a false positive. After the three |03| bytes are confirmed to be at offset five, a PCRE can verify one of the command IDs is present.

```
alert tcp $EXTERNAL_NET 443 -> $HOME_NET any (
    msg:"MALWARE-CNC Win.Trojan.BugSleep command attempt";
# From C2 Host to Victim
    flow:to_client,established;
# A beacon was already seen leaving the network.
    flowbits:isset,bugsleep_beacon;
# Verify 3 higher order bytes of command ID are zero.
    content:"|03 03 03|",depth 3,offset 5,fast_pattern;
# From the start of the buffer, look for beacon reply (any 4 bytes) followed by one of the command IDs.
    pcre:"/.{4}[\x04\x05\x06\x07\x09\x0C\x0D\x0E\x64\x65\x66]/A";
    metadata:impact_flag red,policy max-detect-ips drop,policy security-ips drop;
    classtype:trojan-activity;
    sid:1000001;
)
```

Figure 15: Snort rule for detecting BugSleep command sent from C2 server

Sharp edges

Sometimes, we are reminded that Snort can handle or interpret data differently than expected. Conveniently, this sample's traffic was a perfect example and opportunity to peek under the hood and see what Snort sees. Originally, our beacon rule looked like this, trying to catch the encoded forward-slash that is always present in the Computer Name/Username string (encoded as a comma).

```
alert tcp $HOME_NET any -> $EXTERNAL_NET 443 (
    msg:"MALWARE-CNC Win.Trojan.BugSleep outbound communication attempt";

# From Victim to C2 Host
    flow:to_server,established;

# Encoded 00 bytes of beacon payload length
    content:"|fd fd fd|",offset 1,depth 3,fast_pattern;

# / encoded as , always present in "PC/User" string
# Max depth of 255 because higher 3 bytes of length are zero
    content:"|2C|",depth 255;

# Set flowbit for BugSleep beacon
    flowbits:set,bugsleep_beacon;

# Do not alert because rule down the line will look for command.
    flowbits:noalert;
)
```

Figure 16: Beacon rule attempting to catch forward-slash in Computer Name/Username string

Recall that the implant will:

1. Connect to the server
2. Send a string length (4 bytes)
3. Send the PC/User string N bytes
4. Read 4 bytes back to ensure a response
5. Read 4-byte command ID and N command data bytes
6. Start sending command responses

As Snort is reading data over the wire, it is interpreting it and sorting it into different buffers (pkt_data, file_data, js_data, http_*, etc.). In this case, as TCP data is being chunked along the wire, Snort is looking at those individual TCP segments. Only after it has enough data will it flush into the larger "TCP stream" buffer so a rule can parse the entire stream sent from a client or server.

Initially, the get command traffic was alerting while the put command traffic was not. Fortunately, Snort 3 comes with a tracing module to help debug these issues. The buffer option will print out Snort's different buffers as they are filled and rule_eval will trace the rule as it is evaluated. The following screenshots are output from individual runs of Snort against each PCAP. "snort.raw" represents an individual packet, while "snort.stream_tcp" represents a reassembled TCP stream.

At the start of the working GetFile command, the beacon size and data can be seen as two separate packets (Figure 17).

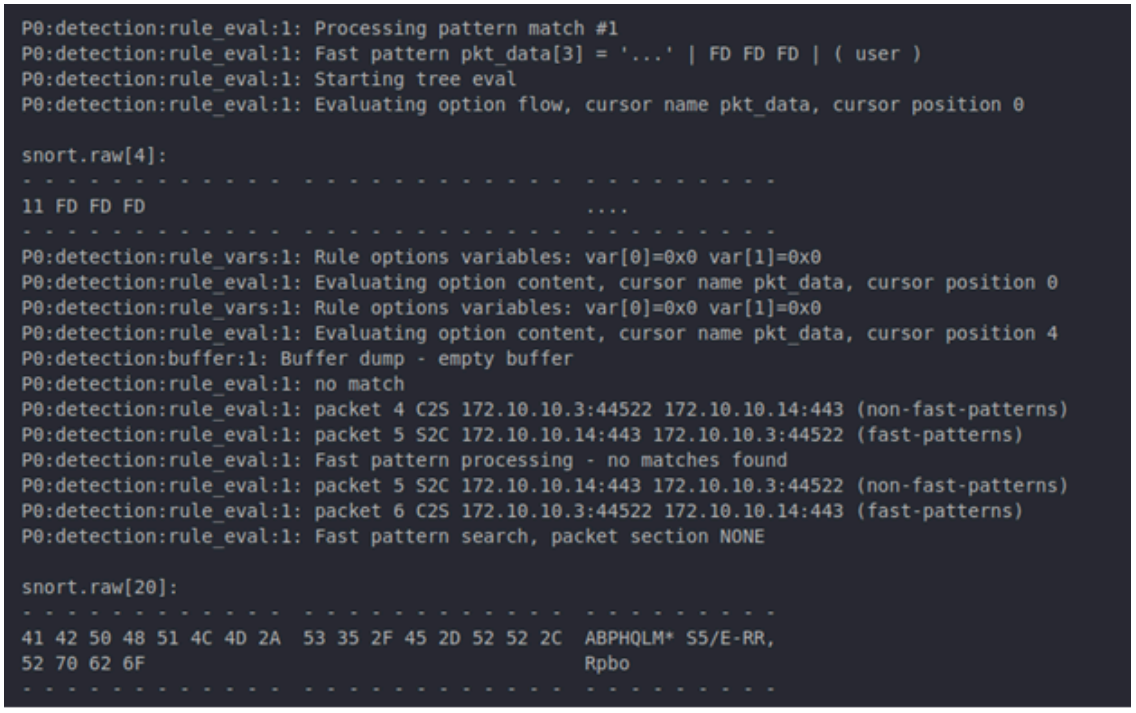


Figure 17: Individual beacon packets being processed by Snort

Further down, the reassembled TCP stream can be seen being inspected and alerted on. Moving from the top to bottom in Figure 18, the cursor position and state of the buffer can be observed changing as the rule is evaluated. At the end, the flowbit is set and made available for the command rule.

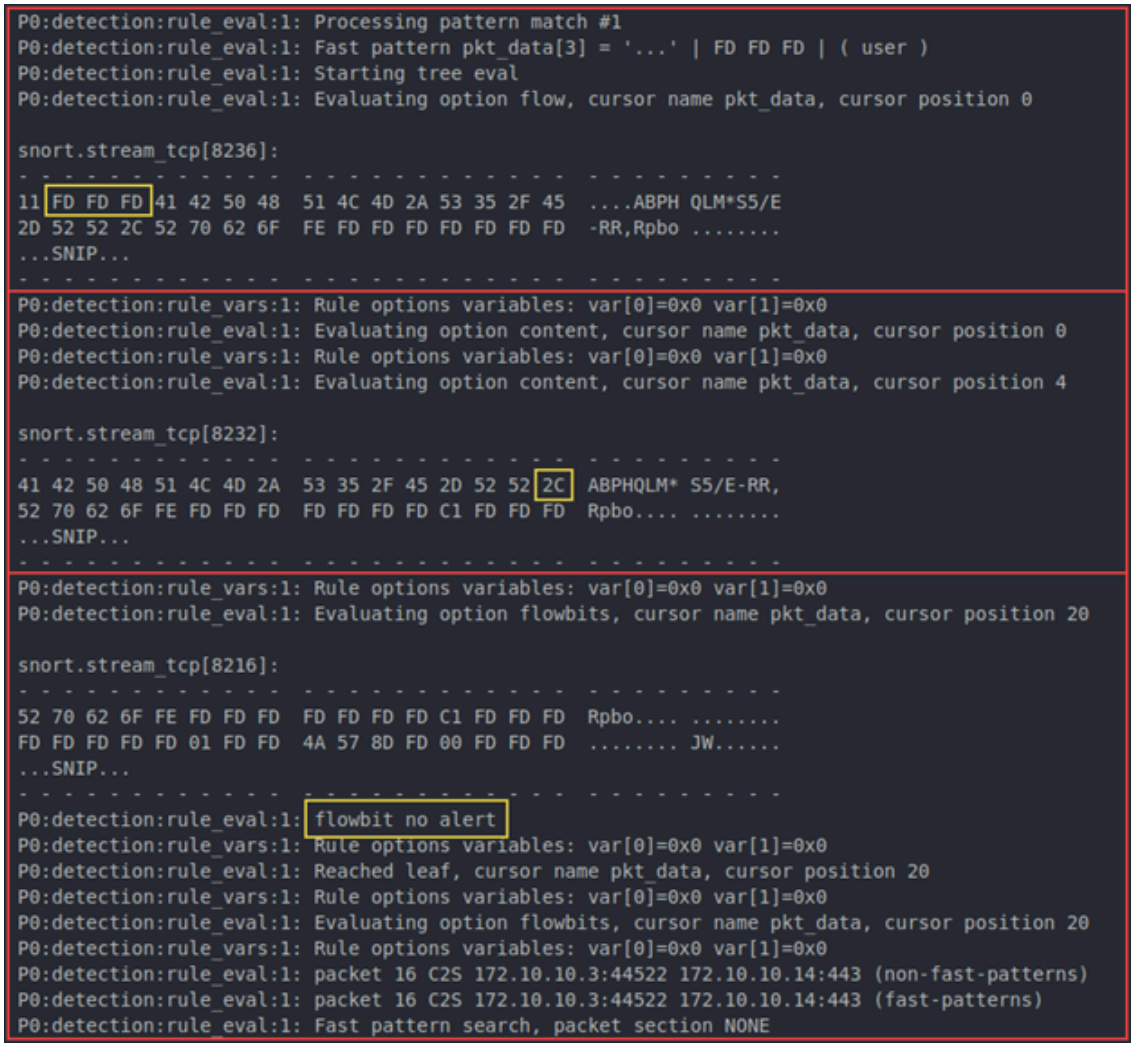


Figure 18: Snort trace output setting flowbit for BugSleep beacon

Further down, the TCP stream for the command data is processed. The higher-order zeroes of the command are found, the flowbit checked, the PCRE performed, and the SID alerts as expected.

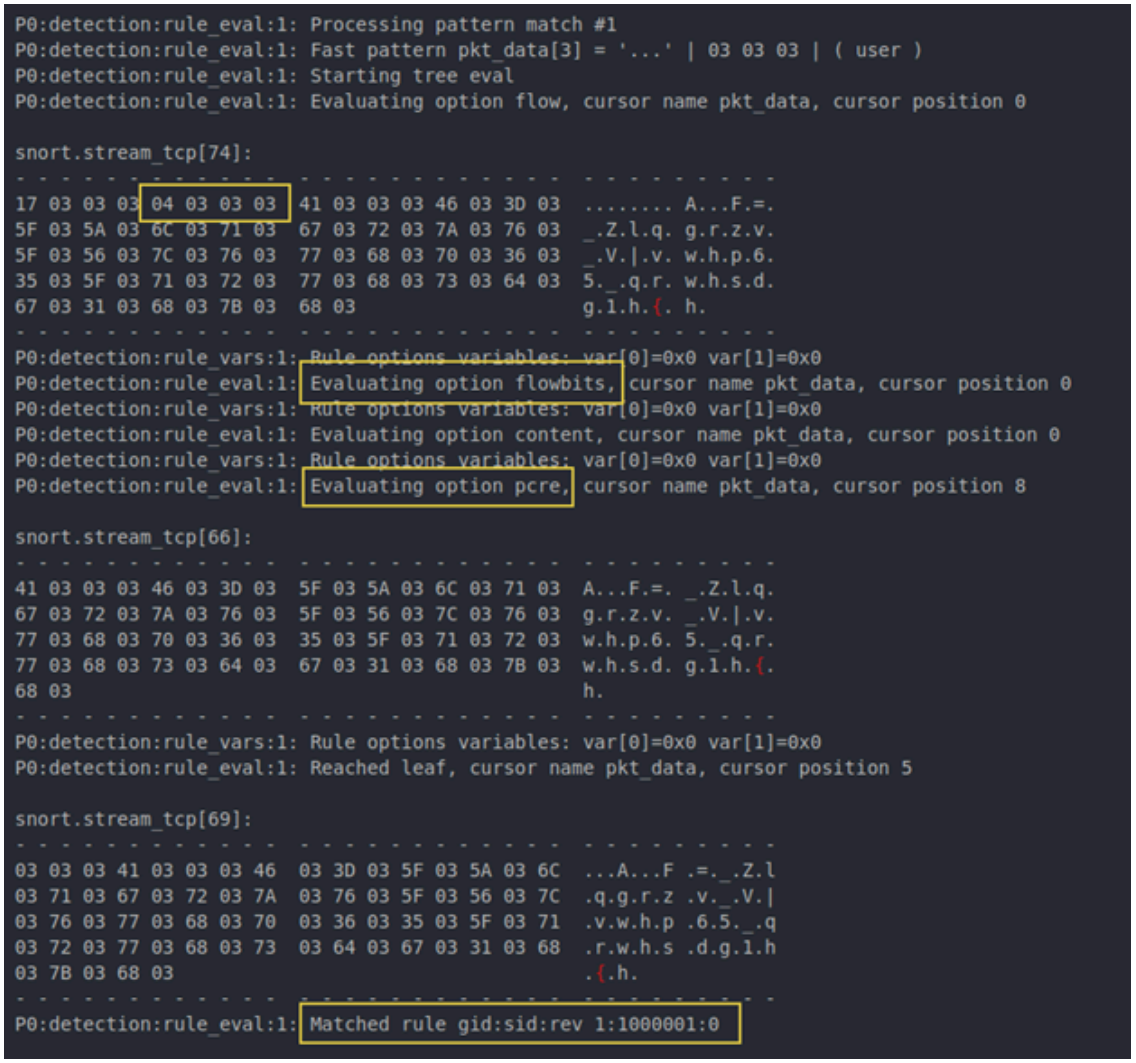


Figure 19: Get file command rule alerts on traffic as expected

When the results of the put file command traffic are inspected, a different behavior is observed. The individual packets for beacon length and beacon data are seen coming in; however, the first reassembled TCP stream that Snort is inspecting is the command being sent back to the implant. Figure 20 shows the command ID being found and then the flowbit check failing.

```
P0:detection:rule_eval:1: Processing pattern match #1
P0:detection:rule_eval:1: Fast pattern pkt_data[3] = '...' | 03 03 03 | ( user )
P0:detection:rule_eval:1: Starting tree eval
P0:detection:rule_eval:1: Evaluating option flow, cursor name pkt_data, cursor position 0

snort.stream_tcp[122]:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
17 03 03 03 05 03 03 03  4B 03 03 03 46 03 3D 03  ........ K...F.=.
5F 03 58 03 76 03 68 03  75 03 76 03 5F 03 58 03  _.X.v.h. u.v._.X.
76 03 68 03 75 03 5F 03  47 03 72 03 66 03 78 03  v.h.u._. G.r.f.x.
70 03 68 03 71 03 77 03  76 03 5F 03 57 03 68 03  p.h.q.w. v._.W.h.
76 03 77 03 49 03 6C 03  6F 03 68 03 31 03 77 03  v.w.I.l. o.h.1.w.
7B 03 77 03 04 03 03 03  21 03 03 03 03 03 03 03  {.w..... !.......
57 6B 6C 76 23 6C 76 23  70 7C 23 77 68 76 77 23  Wklv#lv# p|#whvw#
73 78 77 23 69 6C 6F 68  24 0D                    sxw#iloh $.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
P0:detection:rule_vars:1: Rule options variables: var[0]=0x0 var[1]=0x0
P0:detection:rule_eval:1: Evaluating option flowbits, cursor name pkt_data, cursor position 0
P0:detection:rule_eval:1: failed bit
P0:detection:rule_eval:1: packet 19 S2C 172.10.10.14:443 172.10.10.3:44523 (non-fast-patterns)
P0:detection:rule_eval:1: packet 19 C2S 172.10.10.3:44523 172.10.10.14:443 (fast-patterns)
P0:detection:rule_eval:1: Fast pattern search, packet section NONE
```

Figure 20: Put file command traffic failing flowbit check

Scrolling further in the log reveals the TCP stream for the beacon data is eventually populated and Snort sets the flowbit as expected. The stream for the command ID, however, has already passed and failed analysis because of the unset flowbit, resulting in no alert. The cause of this issue is the raw packets coming from the client not being reassembled into a TCP stream by the time the server packets are reassembled and inspected. This happens because Snort only reassembles when it has enough data, and 20 bytes is not enough yet.

The fix

Unfortunately, the beacon rule must be tweaked so it can alert as soon as possible and not rely on the TCP reassembly. Recall that the beacon function invokes SendSocket twice, once for 4-length bytes and again for the beacon data. This means the first packet Snort sees will only be 4 bytes long. Adding “bufferlen:=4” restricts Snort to only look at 4-byte packets, significantly reducing any FP rate. Our solution ended up being this:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET 443 (
    msg:"MALWARE-CNC Win.Trojan.BugSleep outbound communication attempt";

# Victim to C2 Host
    flow:to_server,established;

# Encoded 00 bytes of beacon payload length
    content:"|fd fd fd|",offset 1,depth 3,fast_pattern;

# Packet will only be 4 bytes long
    bufferlen:=4;

# Set flowbits
    flowbits:set,bugsleep_beacon;
    flowbits:noalert;
)
```

Figure 21: Fixed beacon rule looking for 4-byte length segments

Now the rules work as expected!

```
Alerts
======================
## none-none-Win.Trojan.BugSleep_Command_01_Get-2744081.pcap
     [1:1000001:0]  MALWARE-CNC Win.Trojan.BugSleep command attempt  (1 alerts)

## none-none-Win.Trojan.BugSleep_Command_02_Put-2744081.pcap
     [1:1000001:0]  MALWARE-CNC Win.Trojan.BugSleep command attempt  (1 alerts)

## none-none-Win.Trojan.BugSleep_Command_03_Shell-2744081.pcap
     [1:1000001:0]  MALWARE-CNC Win.Trojan.BugSleep command attempt  (1 alerts)

## none-none-Win.Trojan.BugSleep_Command_04_Inc_Timeout-2744081.pcap
     [1:1000001:0]  MALWARE-CNC Win.Trojan.BugSleep command attempt  (1 alerts)

## none-none-Win.Trojan.BugSleep_Command_06_Kill-2744081.pcap
     [1:1000001:0]  MALWARE-CNC Win.Trojan.BugSleep command attempt  (1 alerts)

## none-none-Win.Trojan.BugSleep_Command_09_Task_Delete-2744081.pcap
     [1:1000001:0]  MALWARE-CNC Win.Trojan.BugSleep command attempt  (1 alerts)

## none-none-Win.Trojan.BugSleep_Command_0A_Task_Check-2744081.pcap
     [1:1000001:0]  MALWARE-CNC Win.Trojan.BugSleep command attempt  (1 alerts)

## none-none-Win.Trojan.BugSleep_Command_0B_Task_Add-2744081.pcap
     [1:1000001:0]  MALWARE-CNC Win.Trojan.BugSleep command attempt  (1 alerts)

## none-none-Win.Trojan.BugSleep_Command_61_Update_Heartbeat-2744081.pcap
     [1:1000001:0]  MALWARE-CNC Win.Trojan.BugSleep command attempt  (1 alerts)

## none-none-Win.Trojan.BugSleep_Command_62_Sock_Timeout-2744081.pcap
     [1:1000001:0]  MALWARE-CNC Win.Trojan.BugSleep command attempt  (1 alerts)

## none-none-Win.Trojan.BugSleep_Command_63_Ping-2744081.pcap
     [1:1000001:0]  MALWARE-CNC Win.Trojan.BugSleep command attempt  (1 alerts)
```

Figure 22: Snort output alerting on traffic from all BugSleep commands

Conclusion

Since BugSleep is a new implant and weekly releases were observed being deployed, this protocol might change and bypass these rules. However, two things have been accomplished:

1. This variant will no longer communicate over our customers' networks.
2. Attackers must invest development time and money to use BugSleep again.

The published Snort SIDs covering this traffic are 63937 and 63938.

Indicators of compromise

IOCs for this research can be found in our GitHub repository here.

Hosts:

- 1[.]235[.]234[.]202
- 146[.]19[.]143[.]14
- 46[.]19[.]143[.]14
- 5[.]239[.]61[.]97

Hashes

The following Windows executables were collected during our research. Assuming these have not been manipulated, the compilation time for this set of binaries indicates weekly releases of BugSleep.

SHA256	Compile Time
b8703744744555ad841f922995cef5dbca11da22565195d05529f5f9095fbfca	Wed., May 8 00:55:53 2024 UTC
94278fa01900fdbfb58d2e373895c045c69c01915edc5349cd6f3e5b7130c472	Wed., May 22 21:56:39 2024 UTC
73c677dd3b264e7eb80e26e78ac9df1dba30915b5ce3b1bc1c83db52b9c6b30e	Fri., May 31 23:29:21 2024 UTC
5df724c220aed7b4878a2a557502a5cefee736406e25ca48ca11a70608f3a1c0	Sun., Jul 07 21:09:49 2024 UTC
960d4c9e79e751be6cad470e4f8e1d3a2b11f76f47597df8619ae41c96ba5809	Sat., Jul 15 09:15:20 2079 UTC
