A Deep Dive Into SoWaT: APT31's Multifunctional Router Implant

Published: 2021-11-25 · Archived: 2026-04-05 19:07:10 UTC

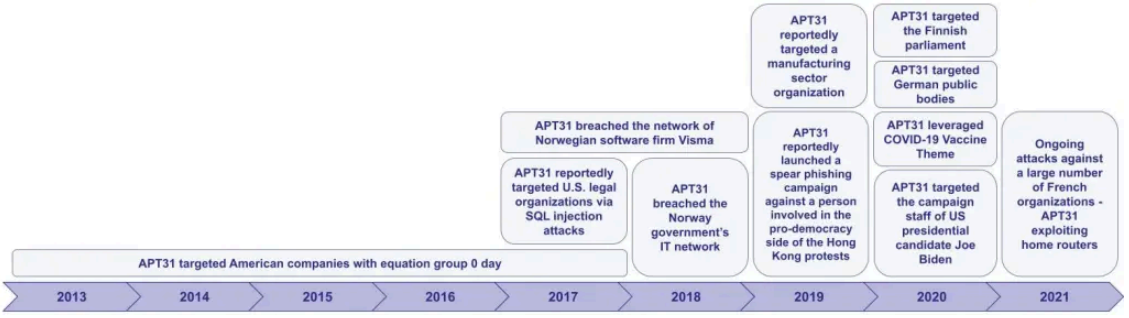
Executive Summary

- APT31 is long known to use Operational Relay Boxes (ORBs) and compromise routers.
- This report examines in detail their only publicly known router implant, dubbed "SoWaT"
- The implant is capable to function as RAT, a tunnel and a proxy.
- Extensive verification and double-encryption procedures signal a TA trying to evade even the most capable defender
- The implant's code reveals a long development history, most likely over several years

Introduction

APT31

APT31, aka Zirconium, is an interesting Chinese threat actor, operating almost separately any other Chinese TA. The group is targeting various types of targets of interest to the Chinese government. Notably, the group has been subject to several governmental attribution statements, including Germany, France, Norway, Australia.



APT31 Timeline, taken with permission from sekoia.io

Sekoia published on the 10/11/21 a great report expanding on previous report by the French ANSSI, The timeline above was copied with permission from them.

It is long known that APT31 is using ORBs (Operational Relay Boxes) for their network operations, specifically hacked routers. Still, there is no public research into how they are operate those ORBs. With this reports I try to shed some light into the groups capabilities on network devices and potential detection and prevention options.

```
time_delta = get_delta_of_time_between_host_internet("time.windows.com");
v5 = *gcs;
gcs[5] = time_delta;
set_curr_time_in_struct(v5, time_delta);
mbedtls_pk_write_pubkey_pem(*(gcs[2] + 4) + 0xBC, *gcs + 0x16, 460);
crc64_hash_460len(*gcs, *gcs + 0x16);
```

SoWaT checking how far off the clock is

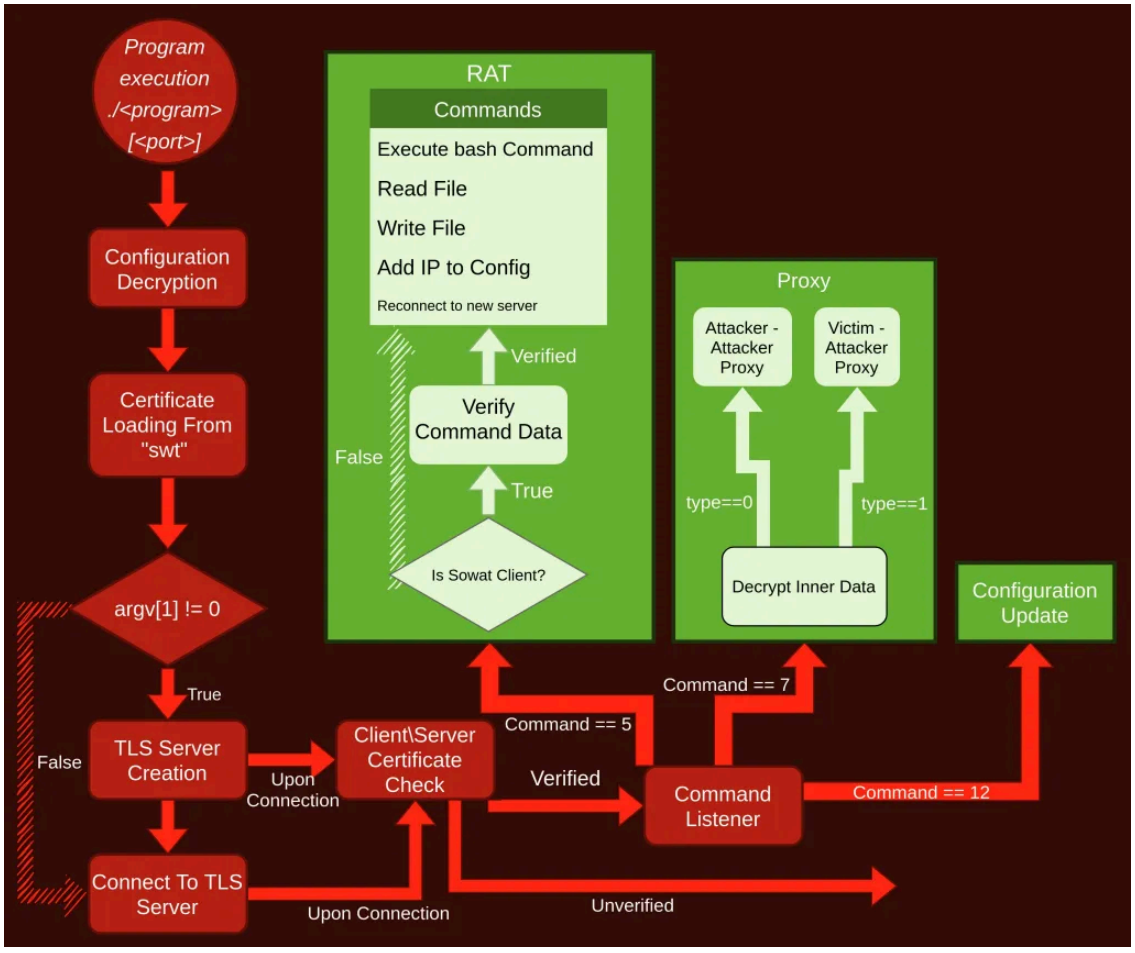
SoWaT

The implant, I dubbed “SoWaT” because of its use of the custom file called “swt”, is an ELF file for 32-bit MIPS based processors. According to the research done by Sekoia, it was most likely used for Pakedge routers, as they represent the vast majority of attacked routers and often run a Linux based OS on MIPS32 processors.

The implant was first mentioned in a tweet by billy leonard as a router implant found by google TAG team and was attributed to APT 31. Sekoia reaffirmed that attribution in their report.

According to the upload time (Nov. 2019) and compilation information of the sample, it was most likely compiled during the fist half of 2019. The compilation used the February 2019 version of cross-compilation tool Buildroot.

Execution Flow



TLS Server and Client

SoWaT communicates with the attacker using TLS, where it can function as a client or as a server, If the implant has a command line argument it start a server on the port given. Regardless of that, it tries to connect to a list of servers specified in its encrypted configuration (See Configuration File) every hour.
Some functions of the implant can only be run if the implant is in server mode.

In both modes, the implant uses certificates and keys parsed from “swt” (See swt). The CA keys to the chain to authenticate the attacker and uses the public and private key for its own.

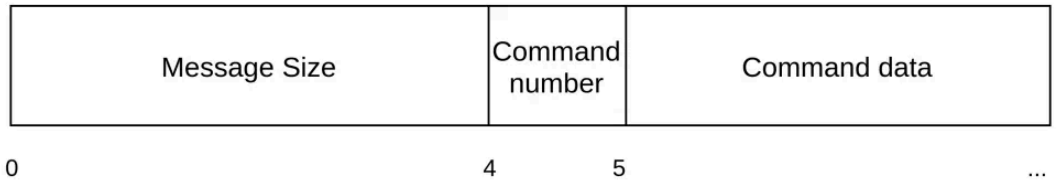
When SoWaT works as server, it sends a Client certificate request, accepting only connections client certificates by the CA from “swt”.
If it attempts to connect as client, it expects the server to send a client certificate request and in response it verifies itself using the private key in “swt”

All components of the TLS connection are implemented by the popular library mbedtls. The specific settings of SoWaT cause the connection to have a unique JA3 and JARM signature, see IOCs.

Command Loop

Interestingly, The main execution function of SoWat isn’t exclusively used to execute command from the attacker but also called from withing the implant itself. This way complicates understanding . Functions in bold are function which were most likely intended to be used by the attacker.

After the TLS connection is successful, SoWaT waits for commands.
Any message needs to have the following basic structure:



Command Message Univerval Structure

All Network protocol structures from here forward are in big endian.

According to the Message size, the implant allocates data and waits for all the data before executing the command. The are 16 possible commands:

Function num.	Description
0	connect to the servers detailed in the configuration
1	Unkown, most likely internal functions
2	Unkown, most likely internal functions
3	Connect to new server with new certificate

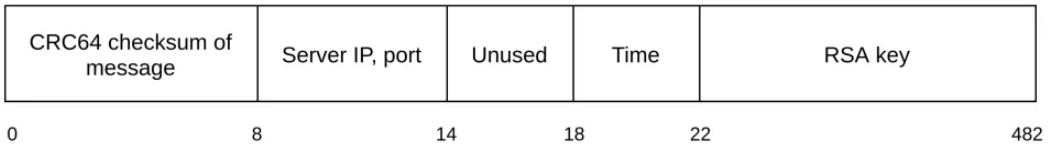
4	Unkown, most likely internal functions
5	start proxy functions
6	query proxy port
7	run RAT
8	reconnect again to Servers in configuration
10	Write current connection IPs to “config” file
11	Get information about all current TLS connections
12	Unkown, likely internal functions
13	Unkown, likely internal functions
15	Update Count of hours without connection
16	Exit process
17	If functions as server, delete all and exit process

Functions in bold are also called by the implant itself.

This command switch place can be confusing, as it combines both function called by the implant for itself and also function that can be called by the attacker. Moreover, some of the function seem to be largely overlapping, most likely because of code added in different parts of the development. The RAT and Proxy functions have their own section, I will describe here other notable functions:

Function 3

The function expects a buffer containing 482-byte long chunks of data detailing a connection to initiate, with the following structure:

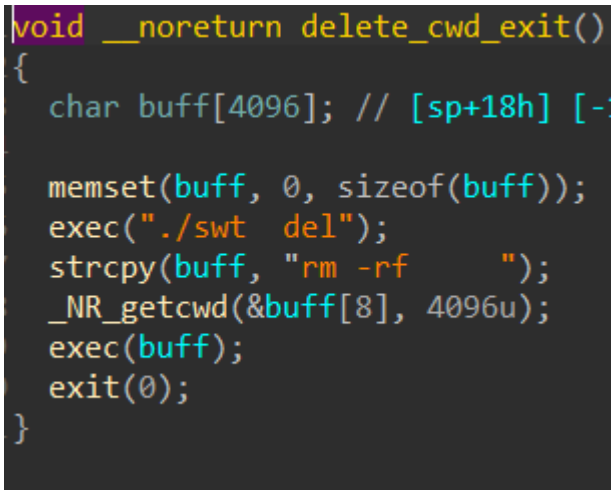


Function 15

The function is called hourly by the implant. It checks whether there were any successfull connections to the implant in the last 24 hours and if there weren’t it calls Function 17. It is unlikely that the function is intended to be called by the attacker directly.

Function 17

This killswitch tries to delete any artifact of the malware: it deletes the entrie current working directory and runs swt to revert any opened ports.

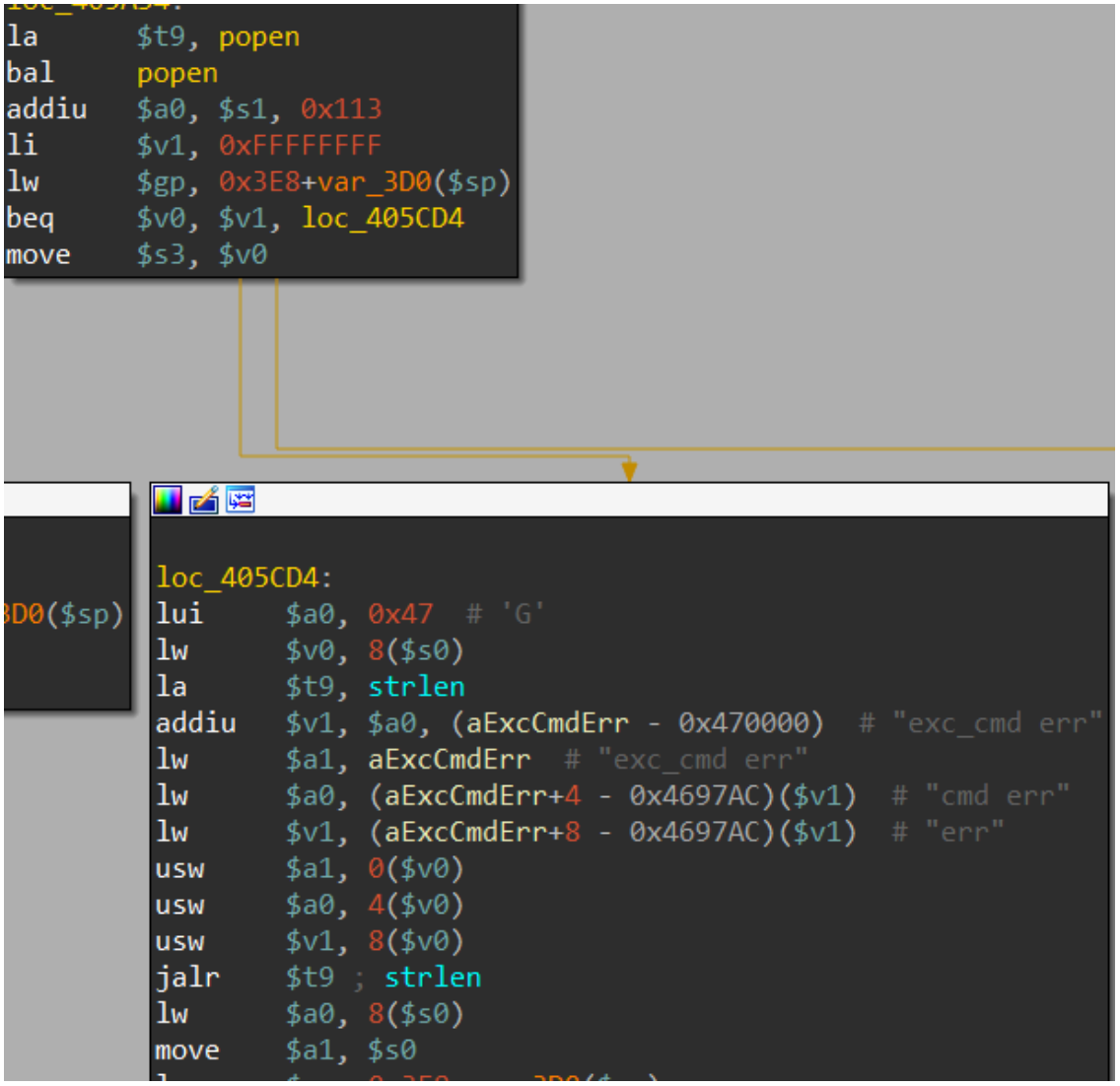


```
void __noreturn delete_cwd_exit()
{
  char buff[4096]; // [sp+18h] [-
  memset(buff, 0, sizeof(buff));
  exec("./swt  del");
  strcpy(buff, "rm -rf     ");
  _NR_getcwd(&buff[8], 4096u);
  exec(buff);
  exit(0);
}
```

RAT

Function num.	explanation
0	Execute command, send output back (“exc_cmd”)
1	Change IP of server
2	Write file “$(CWD)/s”
3, 5	Add IP to configuration and reconnect to that file
4	Delete all files and exit
6	Write to file path chosen by the attacker
7	Add IP to configuration and write file
8	read file

As you can see, the functions are quite simple. Of some interest is function 2, which writes to a file named “s” in the current working directory. I have not seen any other reference to that file

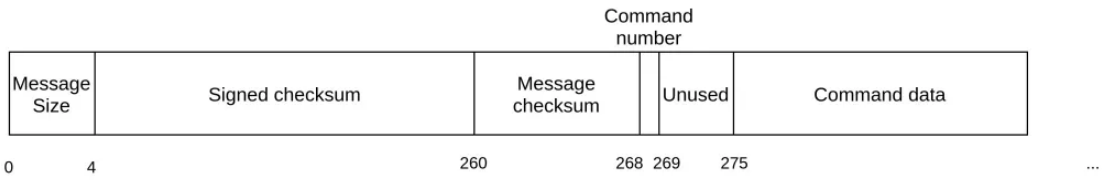


Function number 2

RAT messages

When SoWat receives a RAT message, it first checks if it's the server or the client in the connection. if it's the client it skips the verification, if it functions as a server it does require verification.

This is the message fromat for the verification:



The verification process occurs as follows:

1. Check whether the Signed checksum was signed with the Private key of the Client certificate
2. Calculate the CRC64 Jones checksum of the command data,
3. Compare the message checksum to the CRC64 checksum it calculated.

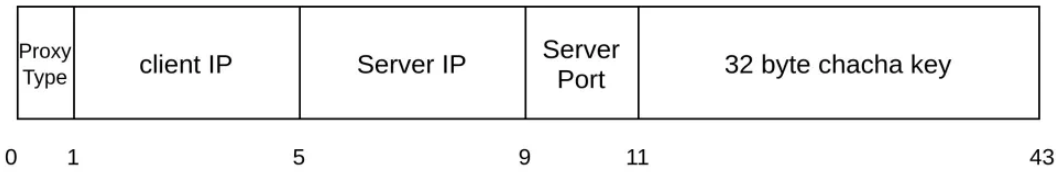
Only if the message is verified the command is ran,

Proxy

SoWaT can function in two different proxy modes:
1. Attacker <-> Attacker
2. Attacker <-> Victim

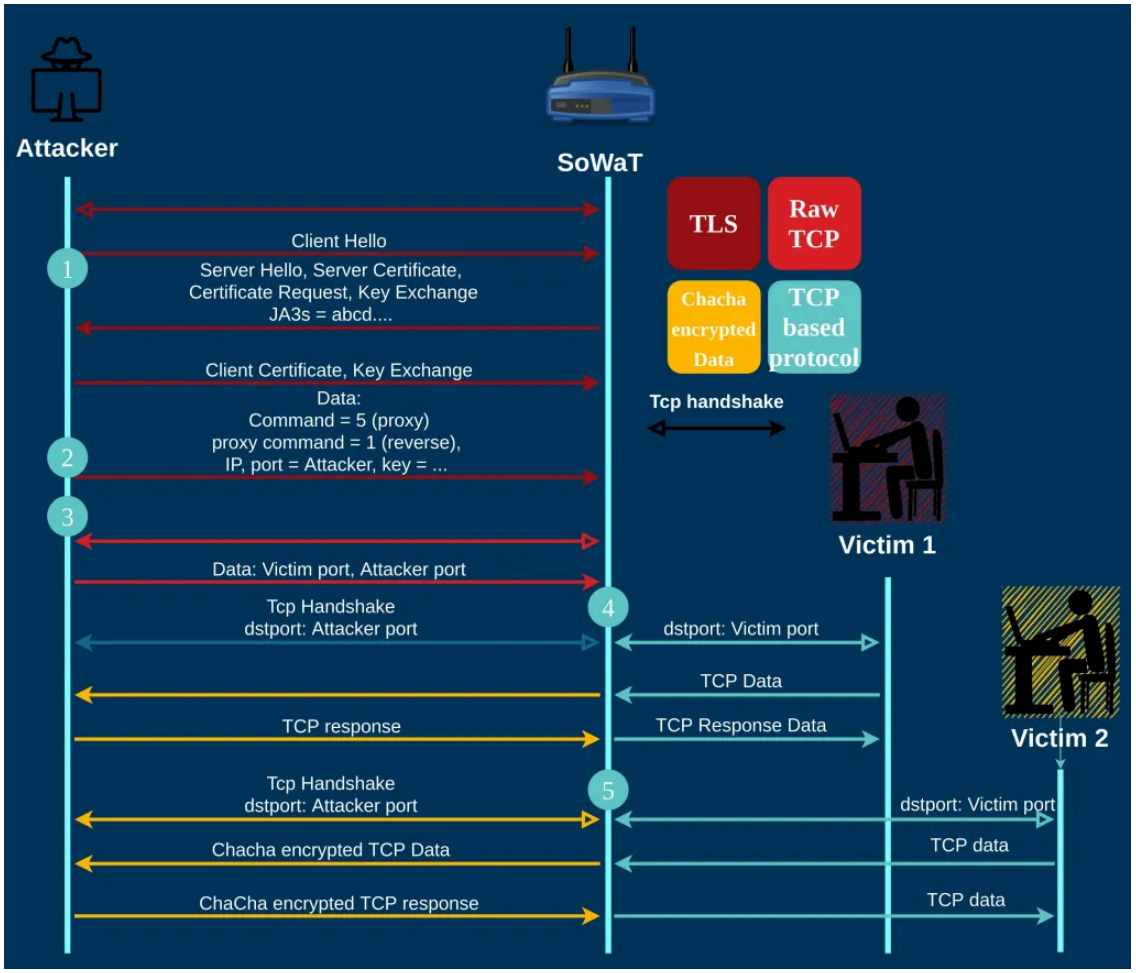
The names of the modes comes only from reversing their internal logic and and analysis of potential uses. While the Attacker <-> Victim mode is quite clear, the Attacker <-> Attacker mode is complicated and it is possible that its purpose is different, though it still works in the way detailed below..
In both modes, the command data needs to to be encrypted using the private key of the client (which it also used to verify itself in the TLS handshake).

The command data, before encryption, is structured in this way:



I have written a script which can connect to SoWaT and perform as both sides of the proxy, See Scripts.

Attacker <-> Victim Proxy

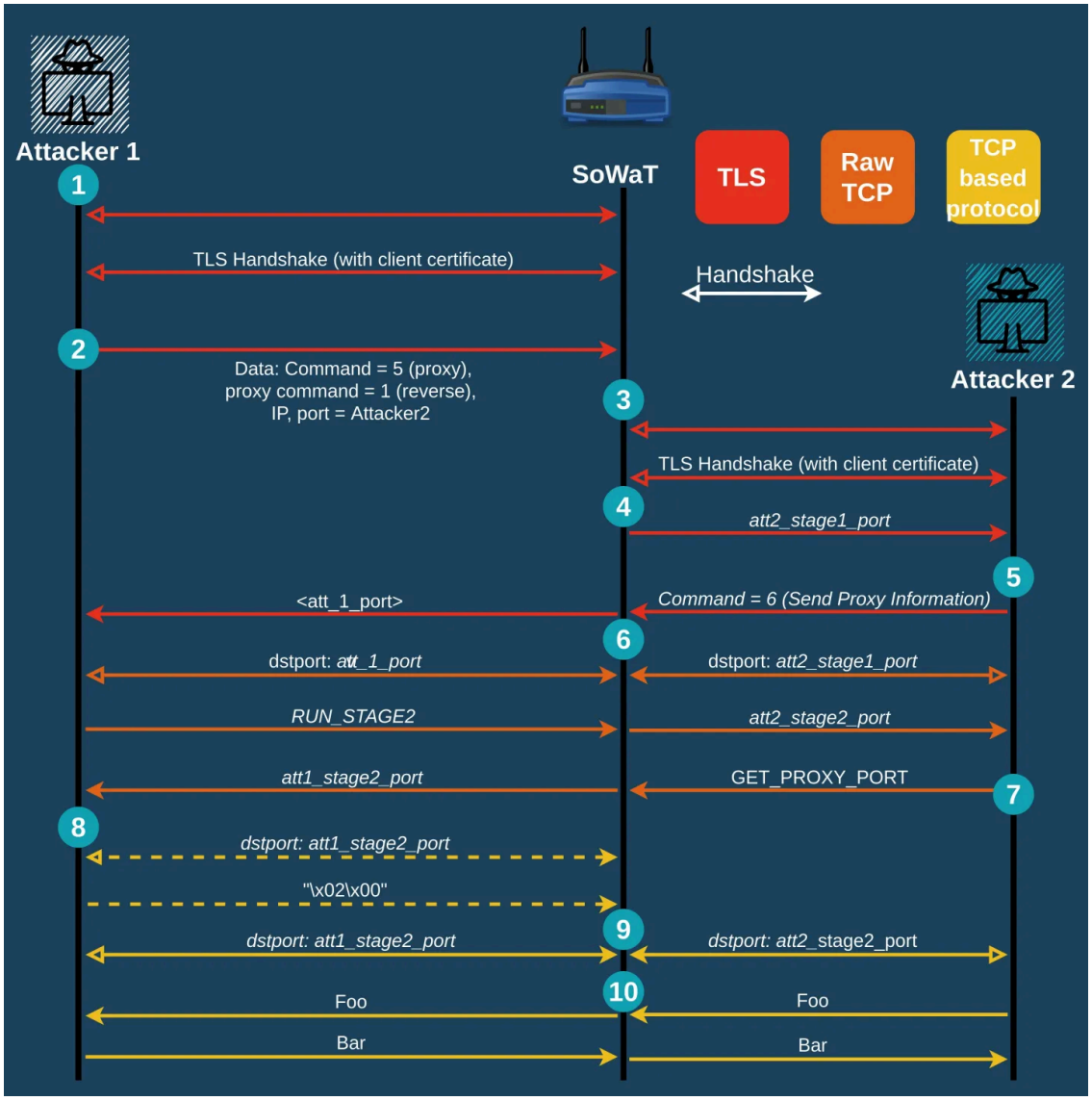


1. The Attacker connects to SoWaT with TLS and client certificate validation
2. The Attacker send a proxy initiation command, with the relevant data (encrypted):
    1. Attacker proxy listening port port and IP
    2. 32 byte ChaCha Key to encrypt communication later.
3. SoWat initiates a raw **unencrypted** TCP socket with the attacker according to the data it got in the proxy command. The attacker sends back the victim connection port and attacker listening port.
4. A malicious program, running on a victim attempts to connect to its server. Its server IP is the SoWaT's address. As soon as it initiates a TCP session with the implant, the implant initiates a **new** TCP session with the attacker.

   The malware encrypts the TCP data from the victim before forwarding it to the attacker and decrypts the data from the attacker before forwarding to the victim. The data is encrypted using the ChaCha Key set by the attacker in step 2.
5. Step 4 can happen simultaneously with up to 5 victims, where each victim prompts the ORB to create a new session with the attacker.

It is important to note that steps 3 onward are not necessarily with the same attacker as steps 1,2. The attacker of steps 3 onward needs only to know at what port to listen to SoWaT.

Attacker <-> Attacker Proxy



1. Attacker 1 (A1) connects to SoWaT with TLS and client certificate validation
2. A1 send a proxy initiation command, with the relevant data (encrypted):
    1. A2 proxy listening port and IP
3. SoWaT connects to Attacker 2 (A2) using TLS and sends its client certificate.
4. SoWaT sends A2 the port for the next stage.
5. A2 sends command 6 to SoWaT, which prompts the implant to send A1 its stage 1 port.
6. Both attackers initiate a TCP connection with SoWat each with the destination ports the implant assigned them
    1. A1 sends command to run stage to the implant, which prompts it to send A2 its stage 2 port
7. A2 sends the ORB a command to send A1 its stage 2 port
8. A1 initiates a TCP connection with its stage 2 port and sends two bytes: {02 00} in order to signal the implant to move forward with this stage.
9. A1 initiates a **new** TCP connection to the same port as in step 8. A2 initiates a TCP connection to the stage2 port it has been given
10. The attackers can communicate with each other through the sockets, and SoWaT servers as a proxy (without encryption).

As In the previous mode, steps 6 onwards can be done by a third attacker and do not need necessarily to continue with A1 which initiaited the proxy command.

But why so complicated?

I do not have a definite answer as to why the proxy intiation process is so complicated, expecially the attacker <-> attacker mode. Two Hyptotheses:

1. **Defense & SIGINT evasion**
    1. Because of the difficult process until finally able to use the proxy, it is much harder for defendersn proactive hunter or anyone wandering in the internet to use it or exfiltrate attackers' data this way.
    2. Finding open ports is becoming irelevant as even if the first port used by the attacker is always the same it is open for a very short time, and the other ports are then random
    3. When having a lot of data, it is harder for some man in the middle to parse every paket to understand what the final proxy port is, which can make it more complicate to capture the relevant attacker traffic.
2. **Compatability and legacy –** It's possible it was easier for the developers to keep in place old code and to just wrap it with their additions, as it does not have any substantial impact on efficiency.

Additional Findings

The Implant File

The implant, available on Virustotal, is an ELF file compiled for the MIPS processor architecture. The ELF is statically linked, with the libraries uClibc, libec, mbedtls, libc-ares and potentially others. While the code is in no matter obfuscated, the stripped binary complicates reversing.

Configuration File

SoWaT assumes a configuration file encrypted using ChaCha20 protocol named "conf" in its current working directory. The file needs to be exactly 386 bytes long, and constructed in this way:

Nonce – Some 12-byte random value which serves as salt for the chacha20 encryption. See ChaCha.
Buffer size – size of the entire structure, in big endian.

The Configuration is decrypted using the following hard-coded key:

```
53 14 3D 23 94 78 A9 68 2F 68 C9 A2 1A 93 3C 5B 39 52 2D 1D E0 63 59 1C 30 44 A2 6A 2A 3F A2 95
```

There are up to 10 IP,Port couples, which represent each an attacker-controlled server the implant may try to connect to.
The structure of the configuration is probably due to compatibility reasons. it is possible that the first 260 bytes were once used to sign the configuration, as that exact buffer length is used in other places for verification.

The “swt” file

The implant depends on a file named “swt” located also in the current working diretory.
I am not in possesion of that file but from examining SoWaT’s use of it is clear It contains a CA certificate, and a signed server certificate and private key. Moreover, it’s also an executable file. From context, it has likely some functionality which opens ports in the router’s firewall or changes its routing so to allow SoWaT to listen to incoming connections in a custom set port.

How SoWaT listens

In order to open a listening socket the implant has a specific procedure:

```
hSocket_1 = _NR_socket(2, 2, 0);
if ( hSocket_1 <= 0 )
  return 0;
hSocket = hSocket_1;
hSocket_FL = fcntl(hSocket_1, F_GETFL, 0);
fcntl(hSocket, F_SETFL, hSocket_FL | 0x80);
sock_opt = 1;
if ( _NR_setsockopt(hSocket, 0xFFFF, 4, &sock_opt) )// SO_REUSEADDR
  return 0;
sockaddr = malloc2_(0x10u);
sockaddr_1 = sockaddr;
if ( !sockaddr )
  return 0;
*sockaddr->sa_data = port;
*&sockaddr->sa_data[6] = 0;
sockaddr->sa_family = AF_INET;
*&sockaddr->sa_data[10] = 0;
*&sockaddr->sa_data[2] = 0;
res_1 = _NR_bind(hSocket, sockaddr, 16);
sockaddr_2 = sockaddr_1;
res = res_1;
free_(sockaddr_2);
if ( res || _NR_listen(hSocket, 5) )
  return 0;
```

Setup of socket

1. It creates a basic socket object (AF_INET, SOCK_DGRAM)
2. It sets the option SO_REUSEADDR to true.
    1. SO_REUSEADDR allows other processes to create another socket listening on the same port.
3. The socket is bound to an open port or to a port directed by the function caller
4. “./swt port <bound port>” is executed as a child process

1. This behavior indicated that the “swt” executable most likely has some ability to alter the firewall or forwarding table of the router.

```
int __fastcall run_swt_port(int port)
{
  char sh_command[260]; // [sp+18h] [-104h] BYREF

  memset(sh_command, 0, 256);
  sprintf_0(sh_command, "./%s  port  %d ", "swt", port);
  return exec(sh_command);
}
```

Dead Code

The implant contains a lot of dead code, which bears resemblance to the malicious code ran. Most likely, this code was used in a previous version of the implant and was included because of compilation flags.

```
MACRO_SIGNAL __fastcall write_signal_to_log(int sig_num)
{
  MACRO_SIGNAL result; // $v0

  if ( sig_num == SIGKILL )
    return writeToLog_("killedd");
  result = SIGINT;
  if ( sig_num == SIGSTKFLT )
  {
    writeToLog_("malloc err");
    result = exec("reboot");
  }
  else if ( sig_num == SIGINT )
  {
    result = writeToLog_("ctrl-c");
  }
  return result;
}
```

Unused Logging function

The implants’ function and architecture show a comprehensive and advanced codebase, relying on many custom structs, and running different tasks simultaneously.

The implant contains very few custom strings, which is quite unusual in router malware, whose developers are less afraid of detection and analysis.
The appearance of strings in an unsystematic way, and the use of status codes may hint that the threat actor has been detected and analyzed in the past and changed their implant to avoid detection.

Execution Path

The implant doesn't give any direct hint to how it was executed. Upon deeper inspection, the string "Usage : ntpclient destination" stands out. which is referenced by a basic print_usage function. This function is never called. Most likely, it is an artifact left by accident by the Threat actor.
The executable lacks the functionality of every ntpclient, i.e. setting the time on the machine, so it was not possible for the implant to masquerade as a that.
It is possible that previous versions of the binary replaced the original ntpclient and so would be started as a daemon, running ntpclient functionality along the malicious functionality. Alternatively, is is possible that the executable was just named ntpclient and the Usage was there to try make it look legitimate to a suspecting IT technician.

```
int print_usage()
{
  return fwrite("Usage : ntpclient destination\n", 1, 30, stderr);
}
```

In the directory the implant is run from 2 files are expected:
"swt"
"conf"

When a kill command is sent, the malware deletes its entrie current working directory, this means that it's likely the attacker dedicate an empty or new folder for their implant

Detection

1. Check for processes with suspicious names or running from unusual paths.
2. I have written YARA rules to detect both unique string and code of the implant, available on my github

AV Detection

As of the time of this writing, the original implant sample was detected by 33 out of 61 AV engines in VT.

Uploading the same sample, with some strings deleted, but fully functional yielded very different results: only 6 AV vendors detected most of these samples.
Essentialy, this means all but the 6 AV vendors are highly unlikely to detect the next SoWaT implant if uploaded to VT.

num.	1	2	5	4	5
detection	6	8	6	6	18
change	changed most strings	removed only unused strings	removed all unique strings	removed all unique strings	change file hash by appending some bytes

				and encryption key	
link	link	link	link	link	link

The 6 AVs that have shown consistent detections are (in alphabetical order):

- AhnLab
- DrWeb
- ESET
- Jiangmin
- Kaspersky
- Zillya

It is important to note that checking if an AV detect a file in VT does not mean whether an AV is good or bad, as in real network AVs detect activity based on many other indicator and not just files.
Nevertheless, this comparison is important in my opinion, especially regarding router malware. it's common that no AV is installed on the network device and to check a suspicious file one can only upload the file to VT or to their own AV portal.

Prevention

The NSA has published in 2020 a great resource detailing how to harden network devices

Conclusions

Development timeline

I can assess with high confidence that SoWaT has had several developmental builds before, for the following reasons:

1. Dead code: Most likely because of a wrong compilation flag the implant contains a lot of dead code, which has a striking resemblance to code that is actually used by the implant. I believe this indicates previous build of the threat actor where they formulated and changed the body of their build.
2. Overlap in functionality: both the RAT and command switcher have function which do almost the same, in a different manner. It is most likely because of backward compatibility.
3. Unnecessary complex code: The code contains a lot of structures and sub-functions which seem to indicate features that were added later to the codebase.

Targets

It is likely that SoWaT is also deployed to other router vendors or linux based servers:

1. The implant expects all incoming communication to be formatted as big endian. This does seem a little strange considering that most likely the attacker is also working on a little-endian based computer. It's

possible that the attacker used that for comfort reasons, but another reason could be that the communication protocols were initially written for a big-endian devices.
2. The implant is fully system-independent and could be run on any linux kernel. The OS dependent part is most likely in the “swt” binary, which changes the firewall of the specific device. The separation between OS-dependent and OS-independent executables does hint that the independent executable is also ran elsewhere.

Verification and Encryption

Throughout the research of the sample it is clear that the developers paid much attention to encryption and verification. Several remarks on that:

1. The implant does not use AES at all and uses only ChaCha for symmetric encryption. That is not only true for the configuration and proxy, but also its TLS supports only one ciphersuite (TLS_DCDHE_RSA_WITH_CHACHA20_POLY1305_SHA256). Most likely, it means the developers deliberately compiled the executable without any AES support. This could be for reasons of efficiency (ChaCha tends to be faster), but also possible that they trust ChaCha to be more secure than AES.
2. The usage of verification and encryption even though TLS is used cannot be explained as an old procedure used before the developers used TLS, because the encryption and verification rely on Client certificate only available throught the TLS handshake. Two other possible explanations are:
    1. The attackers do not trust the TLS protocol so they try additional security measures
    2. Previously a 3rd party successfully did a MiTM attack using a cracked or stolen attacker CA certificate stolen from the attackers. encrypting or verfying data with the client certificate private key makes it difficult for the 3rd party to do that again.

Summary

1. While certainly far from the only router implant, SoWaT underlines that threat actors move to commercially available tools (such as cobalt strike) where it fits them, but do not stop in-house development and research where needed. The fact that the implant was likely developed over multiple years underlines that it is likely not to be thrown away easily.
2. Routers are detection and prevention-wise almost a black hole. The fact that the compromised routers operate as ORBs and most of the time do not harm the organization to whom the router belongs makes prevention and detection even more complicated. Shedding more light and discourse into these may help in developing and distributing more detection and prevention options.
3. This research is far from complete. Any researchers with questions or information are welcome to contact me, and I’ll be happy to share the IDA database and any other relevant data.

APPENDIX A – IOCs

SHA256

1d60edb577641ce47dc2a8299f8b7f878e37120b192655aaf80d1cde5ee482d2 (VT)

File names

swt

s

conf

log

JA3 – e1f787e2b6ec64bf23157522c2e8073d
JARM – 3fd3fd0003fd3fd0003fd3fd3fd3fd755a2cec4b52fb1bce1ac7f1e48c8a7d

APPENDIX B – Scripts

- Proxy Communication Simulation
- Configuration creator
- Basic Openssl certificates creator
- Mock swt creator (depends on certificates)

APPENDIX C – Internal Structures

```
1  struct sowat_info{
2      DWORD current command_num;
3      DWORD calling_method;
4      union message{
5          * char client_response;
6          * char server_response;
7      };
8      union message_len{
9          DWORD client_response_len;
10         DWORD server_response_len;
11     };
12     char client_public_key[460];
13     * global_connection_infromation global_connection_infromation;
14     DWORD unknown;
15     DWORD sock_fd_read;
```

```
16      *   char connection_init_msg;
17      * client_to_server client_to_server;
18      *   void command_run(sowat_info * sowat_info);
19  };
20  struct cert_info
21  {
22      *   void unknown;
23      * mbedtls_x509_crt mbedtls_x509_crt_in_connection;
24      * mbedtls_pk_context mbedtls_pk_context_in_connection;
25  };
26  struct global_connection_infromation
27  {
28      * internal_conn_info internal_conn_info;
29      * DICT known_public_keys;
30      * cert_info cert_info;
31      *   char decrypted_configuration;
32      * ev_loop main_loop;
33      DWORD time_delta;
34      DWORD hours_counter;
35  };
36  struct internal_conn_info
37  {
38      QWORD pk_crc64_jones;
39      DWORD connection_ip;
40      WORD connection_port;
41      DWORD unknown;
```

```
42      DWORD time_of_conn_init;
43      char public_key[460];
44  };
45  struct proxy {
46      BYTE prx_command;
47      DWORD fd_socket_0;
48      DWORD fd_socket_1;
49      DWORD unknown;
50      DWORD tls_server_IP;
51      WORD socket_2_1_port;
52      WORD socket_0_port;
53      WORD socket_1_port;
54      WORD socket_2_0_port;
55      char chacha_encryption_key[32];
56      * ev_io prx_cb_1;
57      * ev_io prx_0_cb0;
58      DWORD fd_socket_prx_1;
59      DWORD time_last_recv_pack1;
60      DWORD time_last_recv_pack2;
61      DWORD time_last_recv_pack3;
62      * ev_loop prx_ev_loop;
63      WORD prx_2_listen_port;
64      DICT used_ports;
65      QWORD unknown;
66      WORD socket_1_port;
67      DICT unknown;
```

```
};
struct DICT{
    void * djb2_hashtable;
    DWORD num_keys;
};
struct dict_hash_obj{
    DWORD djb2;
    *  void value;
    * dict_hash_obj next;
    char key[4];
};
```

Source: https://imp0rtp3.wordpress.com/2021/11/25/sowat/