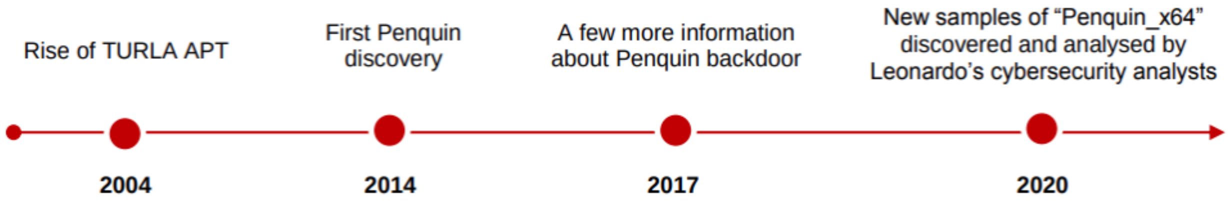
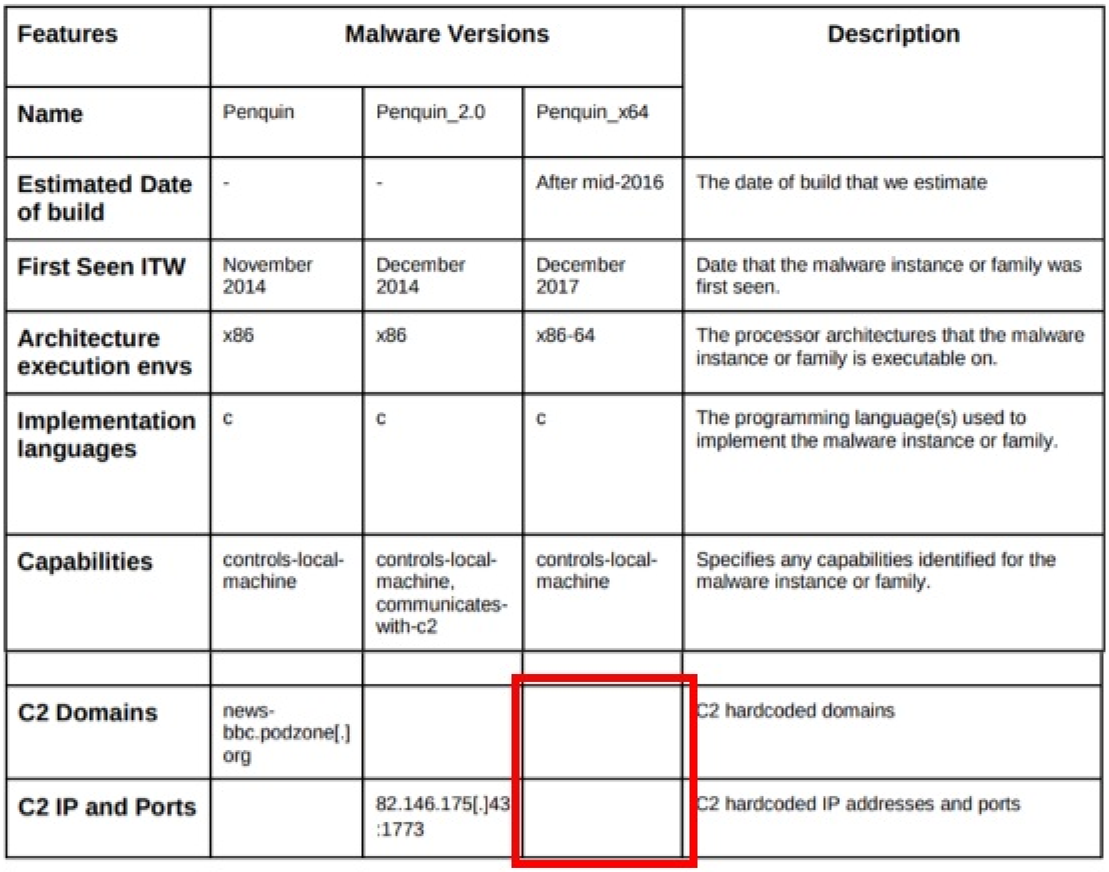
Looking for Penquins in the Wild

lab52.io/blog/looking-for-penquins-in-the-wild/

During 2020 Leonardo analysts discovered and published a very in depth analysis of a threat known as Penquin, attributed to the APT group Turla. 32-bit samples of this threat had been detected and analyzed by Kaspersky before, but the analysis in this most recent publication was focused on a new 64-bit sample.

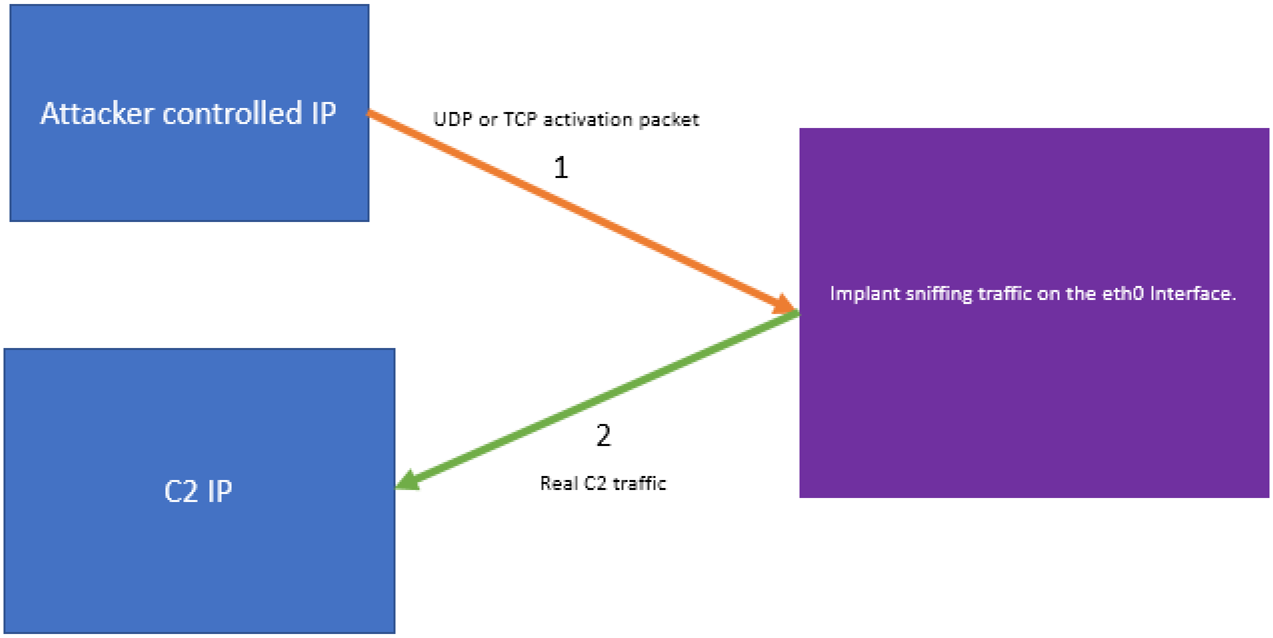

Rise of TURLA APT
2004
First Penquin discovery
2014
A few more information about Penquin backdoor
2017
New samples of "Penquin_x64" discovered and analysed by Leonardo's cybersecurity analysts
2020

It firstly caught our attention the fact that this threat does not have its own command and control server, but rather stands by waiting for a very specific packet generated by the attacker and from that packet it extracts its command and control server.



Features	Malware Versions			Description
Name	Penquin	Penquin_2.0	Penquin_x64	
Estimated Date of build	-	-	After mid-2016	The date of build that we estimate
First Seen ITW	November 2014	December 2014	December 2017	Date that the malware instance or family was first seen.
Architecture execution envs	x86	x86	x86-64	The processor architectures that the malware instance or family is executable on.
Implementation languages	c	c	c	The programming language(s) used to implement the malware instance or family.
Capabilities	controls-local-machine	controls-local-machine, communicates-with-c2	controls-local-machine	Specifies any capabilities identified for the malware instance or family.
C2 Domains	news-bbc.podzone[.]org			C2 hardcoded domains
C2 IP and Ports		82.146.175[.]43:1773		C2 hardcoded IP addresses and ports

This results in the following logical scenario for an attacker to take control of the threat:


Attacker controlled IP
UDP or TCP activation packet
1
Implant sniffing traffic on the eth0 Interface.
2
Real C2 traffic
C2 IP

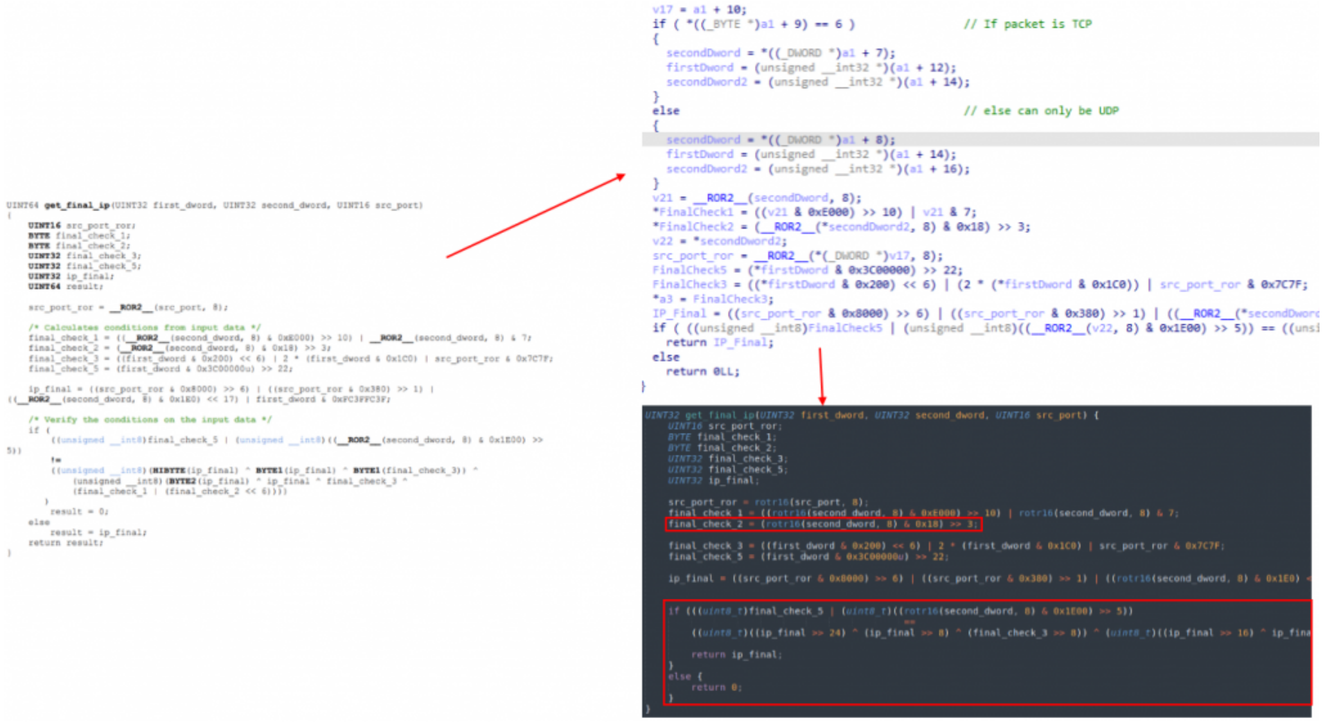
In the report published by Leonardo there is a lot of information related to the structure of such packet and the threat activation protocol, in fact, they published a version of an UDP scanner with a specific packet contacting an internal IP, in order to allow scanning internal networks for threats.

This threat, despite being a compiled and relatively complex binary, has the same capabilities that are usually found in webshells. Furthermore, the group itself could be said to use them in a similar way than other less advanced groups using webshells since it has been observed how in servers infected by this threat, command and controls from other Turla tools have been deployed in order to use them as infrastructure in recent campaigns.

This fact implies that the possibility of detecting new infections of this threat all over the Internet may allow identifying the infrastructure of current, and even future, Turla campaigns.

For this, we firstly need to be able to generate activation packets for public IPs controlled by us. Secondly, we are interested in being able to scan TCP ports as well, since there are cases of servers that have specific TCP ports exposed to the Internet and all UDP ports blocked.

Starting from public samples and the work already done and documented as a resource, we decided to implement a function that emulates the threat’s logic of checking the validity of that “Magic” packet and extracting the IP address of the C2 it is ordered to contact.



```
UINT64 get_final_ip(UINT32 first_dword, UINT32 second_dword, UINT16 src_port)
{
    UINT16 src_port_ror;
    BYTE final_check_1;
    BYTE final_check_2;
    UINT32 final_check_3;
    UINT32 final_check_5;
    UINT32 ip_final;
    UINT64 result;

    src_port_ror = __ROR2__(src_port, 8);

    /* Calculates conditions from input data */
    final_check_1 = ((__ROR2__(second_dword, 8) & 0xE000) >> 10) | __ROR2__(second_dword, 8) & 7;
    final_check_2 = (__ROR2__(second_dword, 8) & 0x18) >> 3;
    final_check_3 = ((first_dword & 0x200) << 6) | 2 * (first_dword & 0x1C0) | src_port_ror & 0x7C7F;
    final_check_5 = (first_dword & 0x3C00000u) >> 22;

    ip_final = ((src_port_ror & 0x8000) >> 6) | ((src_port_ror & 0x380) >> 1) |
((__ROR2__(second_dword, 8) & 0x1E0) << 17) | first_dword & 0xFC3FFC3F;

    /* Verify the conditions on the input data */
    if (
        ((unsigned __int8)final_check_5 | (unsigned __int8)((__ROR2__(second_dword, 8) & 0x1E00) >>
5))
        !=
        ((unsigned __int8)(HIBYTE(ip_final) ^ BYTE1(ip_final) ^ BYTE1(final_check_3)) ^
            (unsigned __int8)(BYTE2(ip_final) ^ ip_final ^ final_check_3 ^
            (final_check_1 | (final_check_2 << 6))))
       )
        result = 0;
    else
        result = ip_final;
    return result;
}
```

```
v17 = a1 + 10;
if ( *((_BYTE *)a1 + 9) == 6 )                  // If packet is TCP
{
  secondDword = *((_DWORD *)a1 + 7);
  firstDword = (unsigned __int32 *)(a1 + 12);
  secondDword2 = (unsigned __int32 *)(a1 + 14);
}
else                                            // else can only be UDP
{
  secondDword = *((_DWORD *)a1 + 8);
  firstDword = (unsigned __int32 *)(a1 + 14);
  secondDword2 = (unsigned __int32 *)(a1 + 16);
}
v21 = __ROR2__(secondDword, 8);
*FinalCheck1 = ((v21 & 0xE000) >> 10) | v21 & 7;
*FinalCheck2 = (__ROR2__(*secondDword2, 8) & 0x18) >> 3;
v22 = *secondDword2;
src_port_ror = __ROR2__(*(_DWORD *)v17, 8);
FinalCheck5 = (*firstDword & 0x3C00000) >> 22;
FinalCheck3 = ((*firstDword & 0x200) << 6) | (2 * (*firstDword & 0x1C0)) | src_port_ror & 0x7C7F;
*a3 = FinalCheck3;
IP_Final = ((src_port_ror & 0x8000) >> 6) | ((src_port_ror & 0x380) >> 1) | ((__ROR2__(*secondDword
if ( ((unsigned __int8)FinalCheck5 | (unsigned __int8)((__ROR2__(v22, 8) & 0x1E00) >> 5)) == ((unsi
  return IP_Final;
else
  return 0LL;
}
```

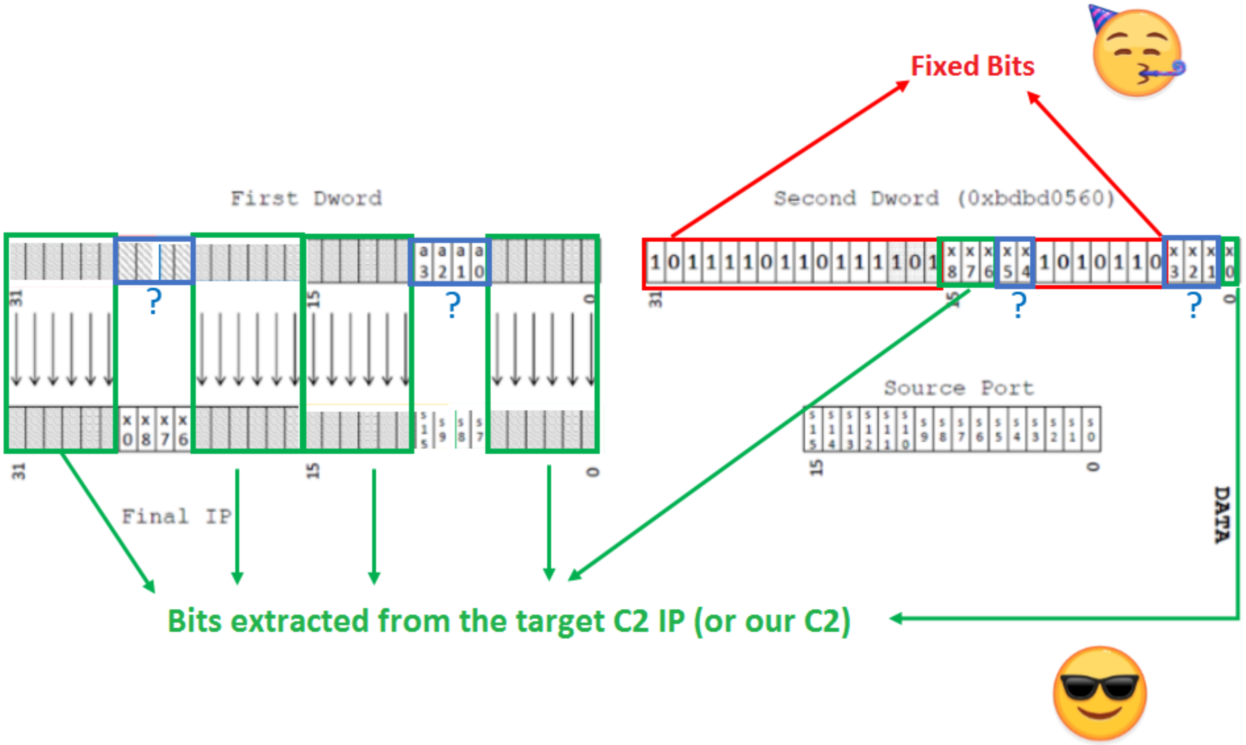
```
UINT32 get_final_ip(UINT32 first_dword, UINT32 second_dword, UINT16 src_port) {
    UINT16 src_port_ror;
    BYTE final_check_1;
    BYTE final_check_2;
    UINT32 final_check_3;
    UINT32 final_check_5;
    UINT32 ip_final;

    src_port_ror = rotr16(src_port, 8);
    final_check_1 = ((rotr16(second_dword, 8) & 0xE000) >> 10) | rotr16(second_dword, 8) & 7;
    final_check_2 = (rotr16(second_dword, 8) & 0x18) >> 3;

    final_check_3 = ((first_dword & 0x200) << 6) | 2 * (first_dword & 0x1C0) | src_port_ror & 0x7C7F;
    final_check_5 = (first_dword & 0x3C00000u) >> 22;

    ip_final = ((src_port_ror & 0x8000) >> 6) | ((src_port_ror & 0x380) >> 1) | ((rotr16(second_dword, 8) & 0x1E0) <

    if (((uint8_t)final_check_5 | (uint8_t)((rotr16(second_dword, 8) & 0x1E00) >> 5))
                                                ==
        ((uint8_t)((ip_final >> 24) ^ (ip_final >> 8) ^ (final_check_3 >> 8)) ^ (uint8_t)((ip_final >> 16) ^ ip_fina

        return ip_final;
    }
    else {
        return 0;
    }
}
```

This allows us to quickly and automatically validate an inferred activation packet.

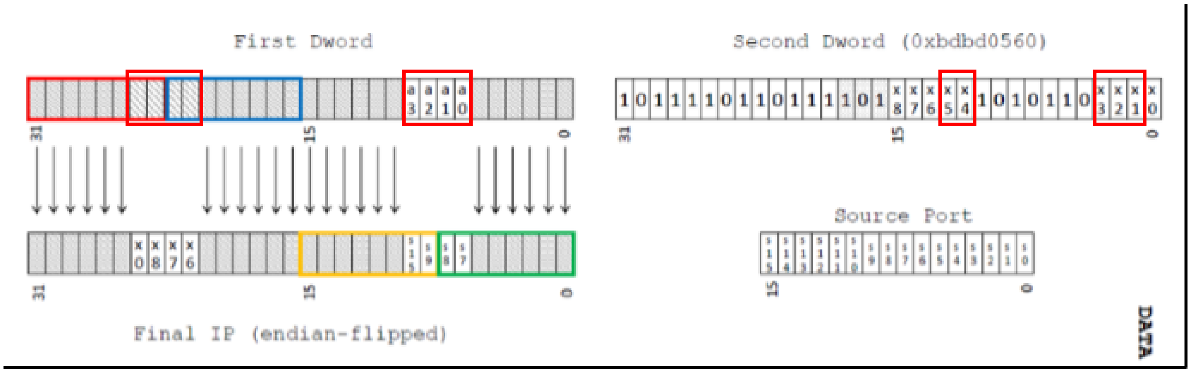
Rather than completely reverse the validation logic, in order to reverse the algorithm and try to generate specific packets we first tried brute-forcing the different elements within the UDP and TCP packets, and then leverage the function we had extracted from the threat to validate each combination for a public target IP controlled by us.

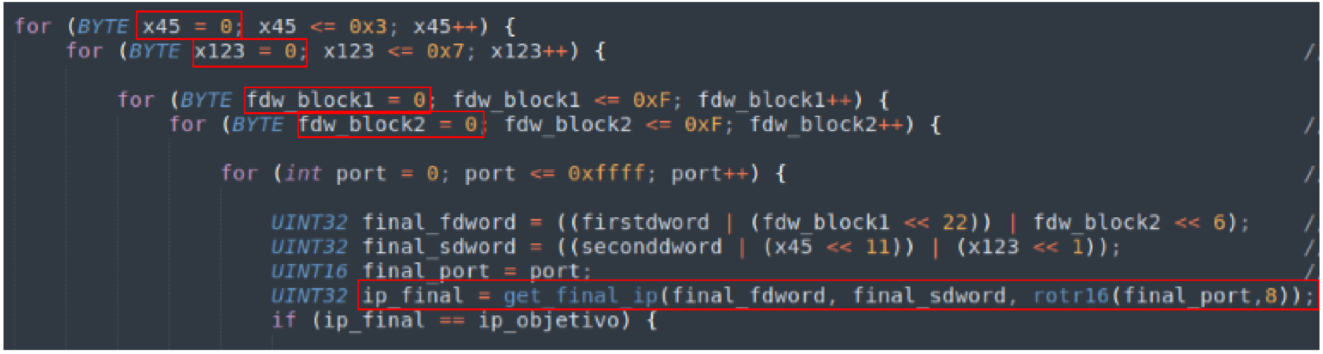
We know that the sample first extracts 32 bits from the packet, and compares them with the mask “0xbdbd0560”. In case it passes this first filter, it extracts another 32 bits and the source port, and passes this extracted information to the validation function we already have. The problem was that brute-forcing so many bytes would be too slow and not feasible.

Fortunately, there is a good part of the algorithm already described. Some of these elements are the fixed bits of the first mask and the bits that compose the IP address with which Penquin will contact (our controlled IP, in this case).


Fixed Bits
First Dword
Second Dword (0xbdbd0560)
1 0 1 1 1 1 0 1 1 0 1 1 1 1 0 1 x8 x7 x6 x5 x4 1 0 1 0 1 1 0 x3 x2 x1 x0
a3 a2 a1 a0
x0 x8 x7 x6
?
31
15
0
Source Port
Final IP
DATA
Bits extracted from the target C2 IP (or our C2)

The fixed bits of the first check take away most of the second block of 32 bits, and since we will want to generate the packets for an IP address controlled by us, we can subtract another 28 bits from the two blocks of 32, since these will have to be exactly those that build the IP of our server. In fact we could subtract 4 more bits from the source port that will also be dependent on the final IP, although in our case we consider it unnecessary. So instead of brute-forcing all possible combinations of two 32-bit blocks plus the port (16 bits more), we only have to brute-force 4 bits + 4 bits + 2 bits + 3 bits + the source port, which becomes trivial.

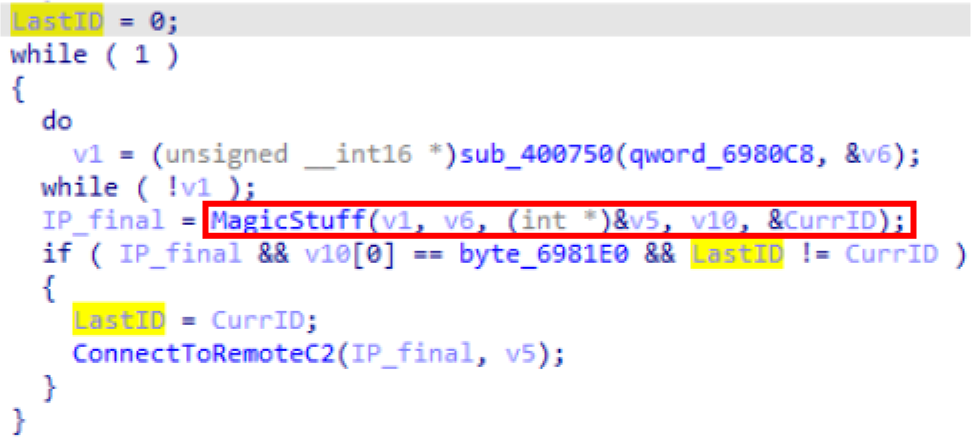

First Dword
Second Dword (0xbdbd0560)
1 0 1 1 1 1 0 1 1 0 1 1 1 1 0 1 x8 x7 x6 x5 x4 1 0 1 0 1 1 0 x3 x2 x1 x0
a3 a2 a1 a0
x0 x8 x7 x6
31
15
0
Source Port
Final IP (endian-flipped)
DATA



```
for (BYTE x45 = 0; x45 <= 0x3; x45++) {
    for (BYTE x123 = 0; x123 <= 0x7; x123++) {                                   /

        for (BYTE fdw_block1 = 0; fdw_block1 <= 0xF; fdw_block1++) {
            for (BYTE fdw_block2 = 0; fdw_block2 <= 0xF; fdw_block2++) {         /

                for (int port = 0; port <= 0xffff; port++) {                     /

                    UINT32 final_fdword = ((firstdword | (fdw_block1 << 22)) | fdw_block2 << 6);    /
                    UINT32 final_sdword = ((seconddword | (x45 << 11)) | (x123 << 1));              /
                    UINT16 final_port = port;                                                       /
                    UINT32 ip_final = get_final_ip(final_fdword, final_sdword, rotr16(final_port,8));
                    if (ip_final == ip_objetivo) {
```

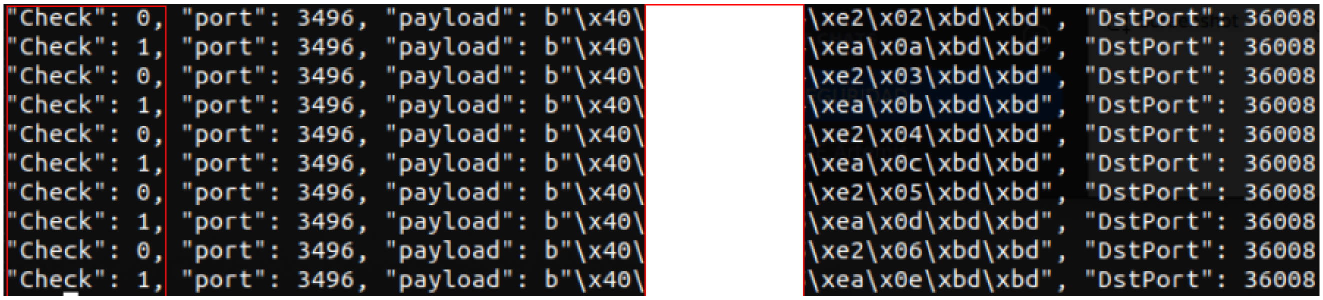
This would allow us to generate combinations for a destination IP and then check if the result is valid and it still generates the IP we expected it to generate.

However, there is still one last element to take into account:



```
LastID = 0;
while ( 1 )
{
  do
    v1 = (unsigned __int16 *)sub_400750(qword_6980C8, &v6);
  while ( !v1 );
  IP_final = MagicStuff(v1, v6, (int *)&v5, v10, &CurrID);
  if ( IP_final && v10[0] == byte_6981E0 && LastID != CurrID )
  {
    LastID = CurrID;
    ConnectToRemoteC2(IP_final, v5);
  }
}
```

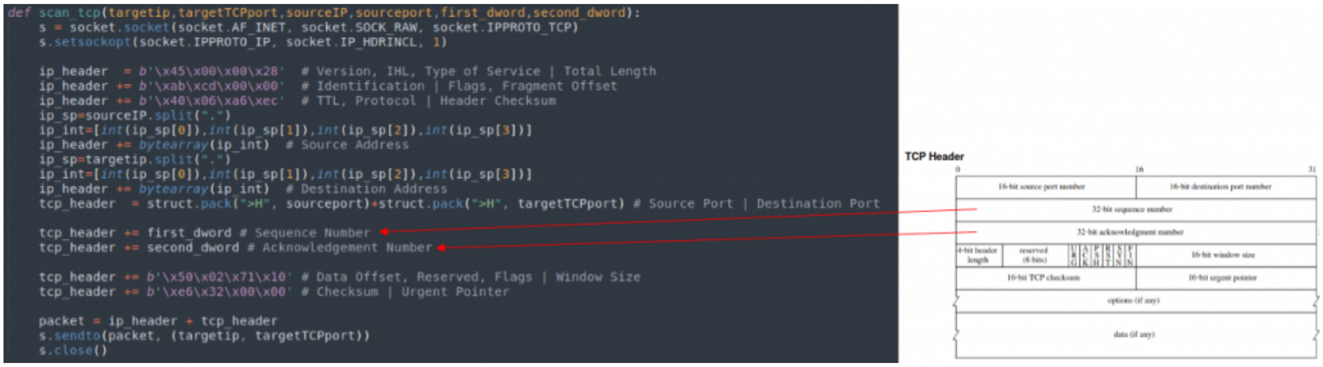
The function that checks the validity of the packet within the threat (in the screenshot renamed to MagicStuff), returns as a parameter an ID extracted from the calculations it performs, which can contain a value between 0 and 3. Just before contacting the target server it compares this ID with the ID of the last contact with a C2, and it is initialized to 0 when the threat is executed for the first time. This avoids accepting the same packet twice, (and at the same time, the first packet cannot result in 0 after that calculation). Therefore, we need to generate two different packets if we want to make sure that in case we send it to an infected server, it will reply and avoid not receiving contact simply because we have sent the same ID as the attacker in the last contact.



```
"Check": 0, "port": 3496, "payload": b"\x40\           \xe2\x02\xbd\xbd", "DstPort": 36008
"Check": 1, "port": 3496, "payload": b"\x40\           \xea\x0a\xbd\xbd", "DstPort": 36008
"Check": 0, "port": 3496, "payload": b"\x40\           \xe2\x03\xbd\xbd", "DstPort": 36008
"Check": 1, "port": 3496, "payload": b"\x40\           \xea\x0b\xbd\xbd", "DstPort": 36008
"Check": 0, "port": 3496, "payload": b"\x40\           \xe2\x04\xbd\xbd", "DstPort": 36008
"Check": 1, "port": 3496, "payload": b"\x40\           \xea\x0c\xbd\xbd", "DstPort": 36008
"Check": 0, "port": 3496, "payload": b"\x40\           \xe2\x05\xbd\xbd", "DstPort": 36008
"Check": 1, "port": 3496, "payload": b"\x40\           \xea\x0d\xbd\xbd", "DstPort": 36008
"Check": 0, "port": 3496, "payload": b"\x40\           \xe2\x06\xbd\xbd", "DstPort": 36008
"Check": 1, "port": 3496, "payload": b"\x40\           \xea\x0e\xbd\xbd", "DstPort": 36008
```

Once two packets with different IDs and a controlled public IP have been generated, the last thing is to send them to a TCP or UDP port that is open on as many servers as possible and wait to receive responses on our server’s IP from the scanned IP addresses.

For this, in the case of UDP we already have the work done, but in the case of TCP the payload that checks the threat is not in the body of the TCP packet received, but in the headers, specifically in the Sequence Number and Aknowledgement Number, so we need to generate the packet in RAW to be able to control these elements:



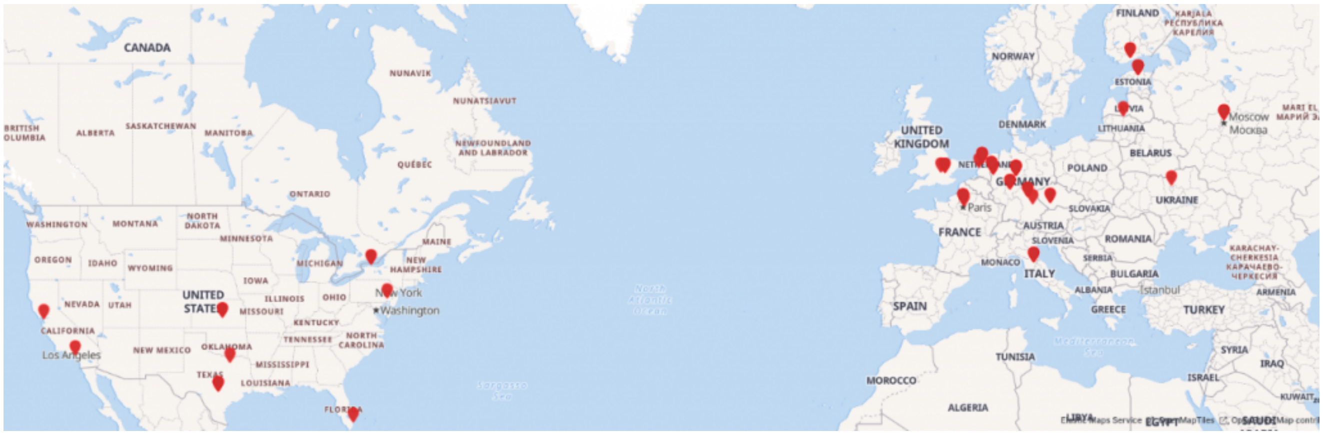
```
def scan_tcp(targetip,targetTCPport,sourceIP,sourceport,first_dword,second_dword):
    s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_TCP)
    s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

    ip_header  = b'\x45\x00\x00\x28'  # Version, IHL, Type of Service | Total Length
    ip_header += b'\xab\xcd\x00\x00'  # Identification | Flags, Fragment Offset
    ip_header += b'\x40\x06\xa6\xec'  # TTL, Protocol | Header Checksum
    ip_sp=sourceIP.split(".")
    ip_int=[int(ip_sp[0]),int(ip_sp[1]),int(ip_sp[2]),int(ip_sp[3])]
    ip_header += bytearray(ip_int)  # Source Address
    ip_sp=targetip.split(".")
    ip_int=[int(ip_sp[0]),int(ip_sp[1]),int(ip_sp[2]),int(ip_sp[3])]
    ip_header += bytearray(ip_int)  # Destination Address
    tcp_header  = struct.pack(">H", sourceport)+struct.pack(">H", targetTCPport) # Source Port | Destination Port

    tcp_header += first_dword # Sequence Number
    tcp_header += second_dword # Acknowledgement Number

    tcp_header += b'\x50\x02\x71\x10' # Data Offset, Reserved, Flags | Window Size
    tcp_header += b'\xe6\x32\x00\x00' # Checksum | Urgent Pointer

    packet = ip_header + tcp_header
    s.sendto(packet, (targetip, targetTCPport))
    s.close()
```

Once all the elements are prepared and finished with a close resemblance to this one:

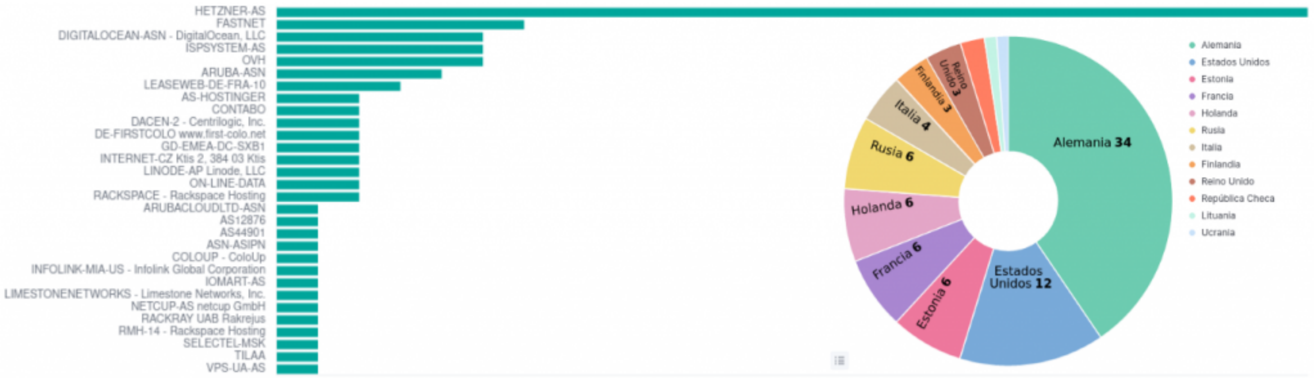
Now we finally can perform the scan 🙂

Different strategies have been used for the scanning. We have scanned ports 25 (TCP), 53 (UDP/ TCP), 80(TCP), 443(TCP), 8080(TCP) and we have tried to vary the “callback” port. To avoid unnecessary traffic, the sending and receiving packets are placed in different ip addresses. The IP address from where it is activated (source of our crafted packets) and the “supposed” C2 (server expected to be contacted) are in different servers. In the case of receiving packets in the expected address, a double check is made sending only to that IP, in order to verify that we are still receiving a response from Penquin.

After a first scan in June 2020 we registered 86 ip addresses hosting Penquin.



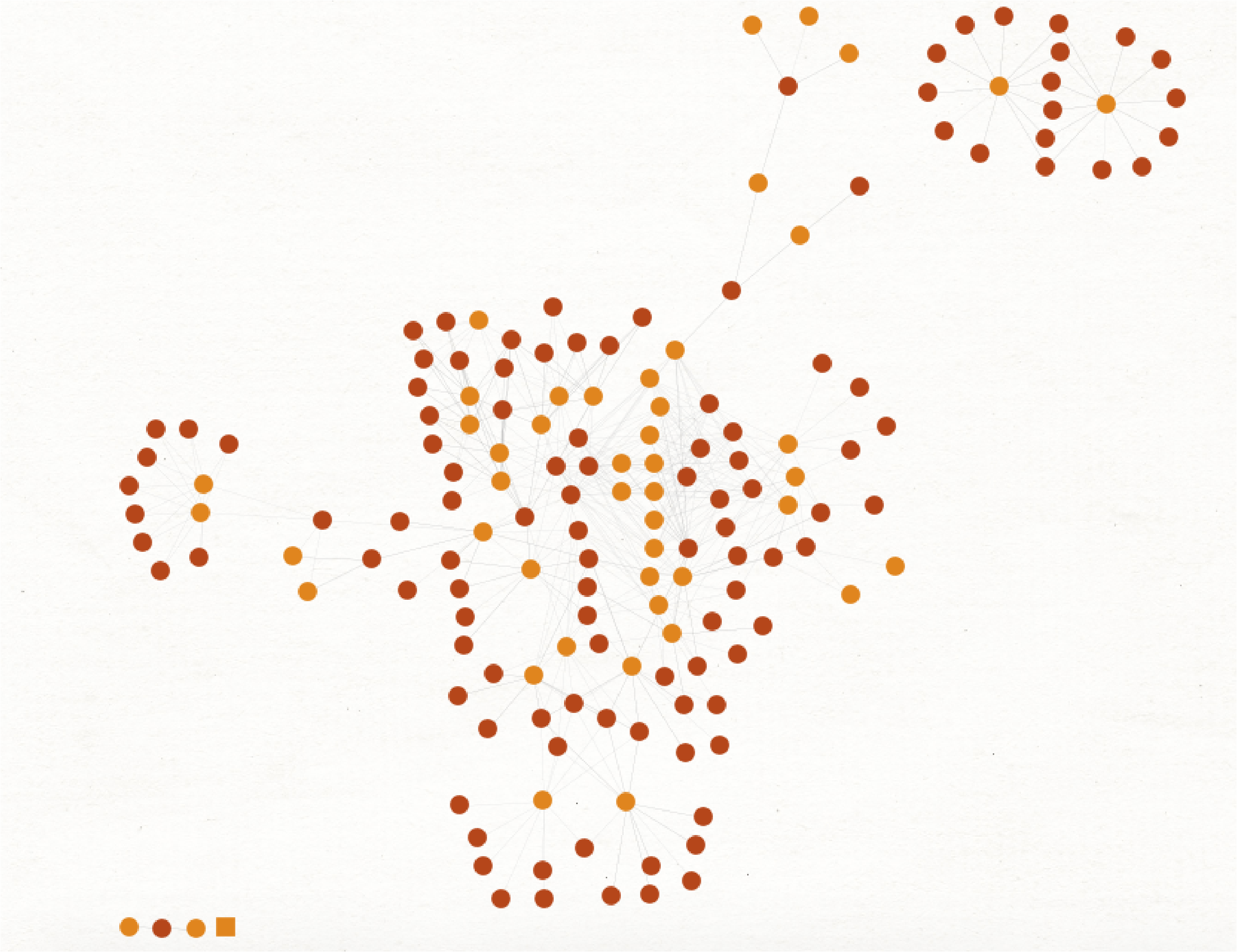
First of all, we were struck by the distribution of these infections, as they were all located in Europe, Russia and the United States.



The IP’s found correspond in most of the cases with VPS from a wide variety of providers.

On the other hand, Shodan offers us information suggesting that many of the IPs, as expected, have multitude of vulnerabilities.

In the image above we can see the vulnerabilities in RED color and the IPs in ORANGE color.

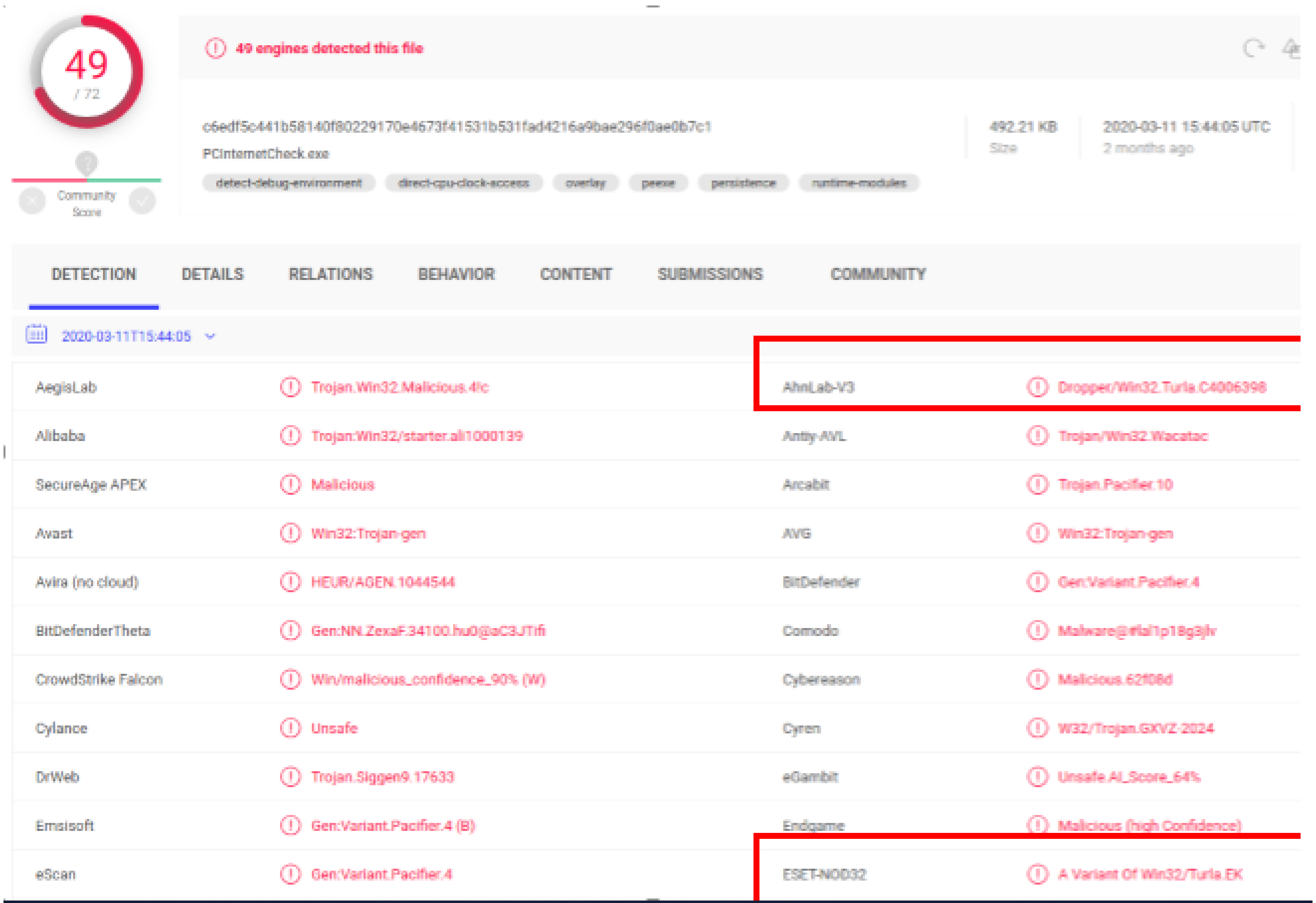


In total, vulnerabilities have been identified in 65% (+ or -) of the detected IPs.

Among the detections observed, these two IP addresses stand out:

- 85.25.95[.]16
- 162.223.94[.]14

At “first sight”, it can be observed how in VT this address is related to a binary that some antivirus vendors along with Intezer catalog as Turla.



49 / 72

Community Score

49 engines detected this file

c6edf5c441b58140f80229170e4673f41531b531fad4216a9bae296f0ae0b7c1

PCInternetCheck.exe

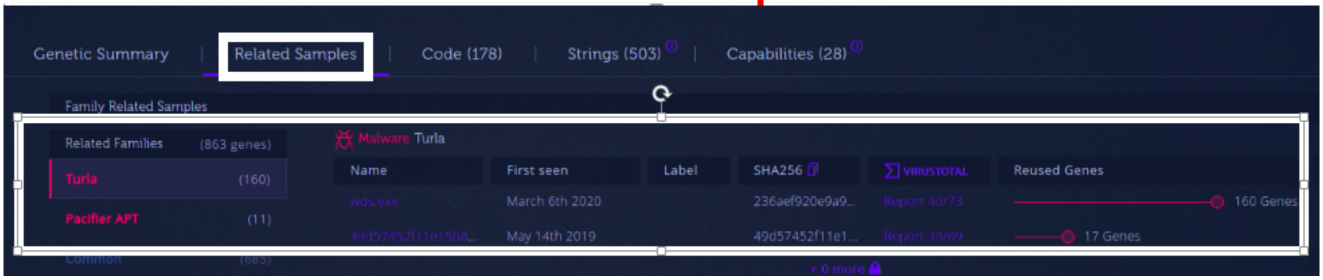
detect-debug-environment · direct-cpu-clock-access · overlay · peexe · persistence · runtime-modules

492.21 KB
Size

2020-03-11 15:44:05 UTC
2 months ago

2020-03-11T15:44:05

AegisLab	Trojan.Win32.Malicious.4!c	AhnLab-V3	Dropper/Win32.Turla.C4006398
Alibaba	Trojan:Win32/starter.ali1000139	Antiy-AVL	Trojan/Win32.Wacatac
SecureAge APEX	Malicious	Arcabit	Trojan.Pacifier.10
Avast	Win32:Trojan-gen	AVG	Win32:Trojan-gen
Avira (no cloud)	HEUR/AGEN.1044544	BitDefender	Gen:Variant.Pacifier.4
BitDefenderTheta	Gen:NN.ZexaF.34100.hu0@aC3JTifi	Comodo	Malware@#fal1p18g3jlv
CrowdStrike Falcon	Win/malicious_confidence_90% (W)	Cybereason	Malicious.62f08d
Cylance	Unsafe	Cyren	W32/Trojan.GXVZ-2024
DrWeb	Trojan.Siggen9.17633	eGambit	Unsafe.AI_Score_64%
Emsisoft	Gen:Variant.Pacifier.4 (B)	Endgame	Malicious (high Confidence)
eScan	Gen:Variant.Pacifier.4	ESET-NOD32	A Variant Of Win32/Turla.EK



Family Related Samples

Related Families	(863 genes)
Turla	(160)
Pacifier APT	(11)
Common	(685)

Malware Turla

Name	First seen	Label	SHA256	VIRUSTOTAL	Reused Genes
wds.exe	March 6th 2020		236aef920e9a9...	Report 46/73	160 Genes
49d57452f11e15b8...	May 14th 2019		49d57452f11e1...	Report 38/69	17 Genes

+ 0 more

Researching for more details about this sample, we find that it is mentioned in an AnhLab Report about a Turla campaign against the Korean Defense sector, which, along with this paragraph from Cisto Talos from July 2021 “One public reason why we attributed this backdoor to Turla is the fact that they used the same infrastructure as they used for other attacks that have been clearly attributed to their Penguin Turla Infrastructure.” Related to their analysis of Tinyturla, reinforces the hypothesis that they use Penquin as a tool to control machines that are then used as command and control servers or intermediate node servers for their operations.

References:

[1] https://www.leonardocompany.com/documents/20142/10868623/Malware+Technical+Insight+_Turla+%E2%80%9CPenquin_x64%E2%80%9D.pdf

[2] https://cn.ahnlab.com/global/upload/download/asecreport/ahnlab_zh_202006%20vol.91.pdf

[3] https://blog.talosintelligence.com/2021/09/tinyturla.html

IOCs

1d5e4466a6c5723cd30caf8b1c3d33d1a3d4c94c25e2ebe186c02b8b41daf905	SHA256
2dabb2c5c04da560a6b56dbaa565d1eab8189d1fa4a85557a22157877065ea08	SHA256
3e138e4e34c6eed3506efc7c805fce19af13bd62aeb35544f81f111e83b5d0d4	SHA256
5a204263cac112318cd162f1c372437abf7f2092902b05e943e8784869629dd8	SHA256
67d9556c695ef6c51abf6fbab17acb3466e3149cf4d20cb64d6d34dc969b6502	SHA256
8856a68d95e4e79301779770a83e3fad8f122b849a9e9e31cfe06bf3418fa667	SHA256
8ccc081d4940c5d8aa6b782c16ed82528c0885bbb08210a8d0a8c519c54215bc	SHA256
d49690ccb82ff9d42d3ee9d7da693fd7d302734562de088e9298413d56b86ed0	SHA256
d9f2467ff11efae921ec83e074e4f8d2eac7881d76bff60a872a801bd45ce3d5	SHA256
85.25.95.16	Infected Server
162.223.94.14	Infected Server

Customers with Lab52's APT intelligence private feed service already have more tools and means of detection for this campaign.
In case of having threat hunting service or being client of S2Grupo CERT, this intelligence has already been applied.

If you need more information about Lab52's private APT intelligence feed service, you can contact us through the following link