28 - 30 September, 2022 / Prague, Czech Republic

TRACKING THE ENTIRE ICEBERG – LONG-TERM APT MALWARE C2 PROTOCOL EMULATION AND SCANNING

Takahiro Haruyama

VMware, Japan

tharuyama@vmware.com

ABSTRACT

Malware analysts normally obtain the IP addresses of malware's command-and-control (C2) servers by analysing samples. This approach works in commoditized attacks or campaigns. However, with targeted attacks using APT malware, it's difficult to acquire a sufficient number of samples for organizations other than anti-virus companies. As a result, malware C2 IOCs collected by a single organization are just the tip of the iceberg.

For years, I have reversed the C2 protocols of high-profile APT malware families then, by emulating the protocols, discovered the active C2 servers on the Internet. In this presentation I will explain how to emulate the protocols of two long-term pieces of malware used by PRC-linked cyber espionage threat actors: Winnti 4.0 and ShadowPad.

Both pieces of malware support multiple C2 protocols, such as TCP / TLS / HTTP / HTTPS / UDP. It's also common to have different data formats and encoding algorithms for each protocol in the same piece of malware. I'll cover the protocol details while referring to unique functions such as server mode in Winnti 4.0 and multiple protocol listening at a single port in ShadowPad. Additionally, I'll share the findings for the Internet-wide C2 scanning.

After the presentation, I will publish over 120 C2 IOCs with the date ranges in which they were discovered. These dates are more helpful than IP address information alone since the C2s are typically found on hosted servers, meaning that the C2 could sometimes exist on a specific IP only for a very limited time. 65% of these IOCs have 0 detection on *VirusTotal* at the time of writing this paper.

INTRODUCTION

Security practitioners often rely on the reputation of IP addresses to determine if traffic to and from that indicator of compromise (IOC) is malicious. However, reputation is not effective for catching new malware C2 servers, especially when the APT actors limit their malware deployments to specific targets. In the example shown in Figure 1, anti-virus (AV) engines detected an IP address as harmless (0/90) on *VirusTotal* in June 2022, but I was able to identify it as a Winnti 4.0 C2 server.

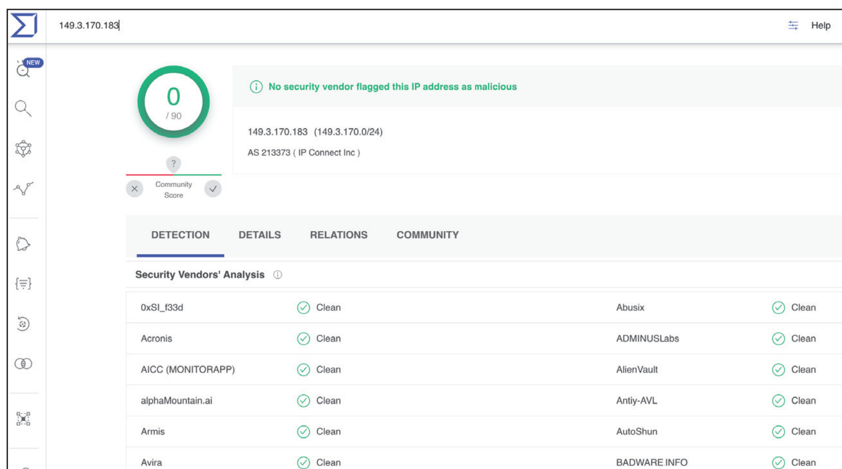

Figure 1: VirusTotal result against one IP address.

I looked at the TLS protocol, which I had reversed, and saw the protocols of high-profile malware families were emulated, in particular those used for cyber espionage to discover real-time C2 instances on the Internet. The following section describes how I created this intelligence for two pieces of long-term APT malware: Winnti 4.0 and ShadowPad.

SELECTION OF LONG-TERM APT MALWARE

Both Winnti 4.0 and ShadowPad are utilized by PRC-linked cyber espionage threat actors. Table 1 shows a summary of each malware as the target of C2 scanning research.

	Winnti 4.0	**ShadowPad**
Prevalence	Low	High
First-observed year	2016 (start-up sequence), 2018 (new C2 protocol)	2015
Scanning start year	2019	2021
Supported protocols	TCP / TLS / HTTP(S) / UDP	TCP / SSL / HTTP(S) / UDP / DNS
Unique feature	Server mode	Multiple protocol listening at a single port

Table 1: Summmary of scanning targets.

There has been minimal reporting on Winnti 4.0 so far, while ShadowPad threat analysis information has been published by many organizations. Multiple C2 protocols are supported by both. I will detail the C2 protocols (especially focusing on the initial handshake), scanner implementations, and the scanning results.

Winnti 4.0

Winnti is a family of malware that has been seen in many large-scale attacks and has been attributed to, at least, the APT41 Chinese threat actor [1]. The malware was reported by *Kaspersky* for the first time in 2013 (version 1.0-2.0) [2]. *Novetta* analysed the start-up sequence and C2 protocol of version 3.0 in detail [3].

I observed that the version 3.0 implementation (in particular the start-up sequence) has changed since 2016, which has guided this research into categorizing Winnti as multiple variants. *Macnica Networks* first described the new implementation at JSAC 2018 [4]. Here we refer to the new variants as version 4.0 to differentiate from previous versions. The differences from version 3.0 are shown in Table 2.

	Version 3.0	**Version 4.0**
Initial component	Dropper	Loader and DAT file
Initial encryption algorithm	DES	AES
Initial encryption key cracking	Easy	Hard
Worker encryption	1-byte XOR and nibble swap	DPAPI or AES with host-specific key

Table 2: Difference between Winnti versions 3.0 and 4.0.

Winnti 3.0 was composed of multiple components: Dropper, Service and Worker. The Dropper component initially decrypts and saves both the Service component and the Worker component then sets the persistence for Service. After that, the Service component runs the main payload, called Worker. Version 4.0 is more carefully implemented to make code acquisition for the Worker component harder. As a result, there has been limited information publicly available about this variant since it was first observed. If the reader is interested in the version 4.0 installation behaviour, check the blog post at [5].

In the process of analysing the Winnti 4.0 samples, I identified the new Worker component from 2018 whose code has less than 50% similarity with the existing version 3.0. The Worker configuration block structure is detailed below:

```
struct __unaligned __declspec(align(2)) struc_work_config
{
   char campaignID[64];
   char MAC_addr[6];
   int c2_proto;
   char c2_host_port[128];
   char c2_active_table[672];
   struc_proxy proxy;
   struc_server_ports server_ports;
   int httpapi_mode_imm4;
   int httpapi_proto;
   int httpapi_port;
   int field_56A;
   int field_56E;
   int field_572;
   int field_576;
   int field_57A;
   int field_57E;
   wchar_t httpapi_url[256];
   char field_782[16];
   int server_cert_size;
   int server_privkey_size;
   char server_cert_der[655]; // flexible according to the size
   char server_privkey_der[609]; // flexible according to the size
   char padding[2832];
};

struct struc_proxy
{
   int proxy_proto;
   char proxy_host[256];
```

```
    char proxy_user[64];
    char proxy_pass[32];
    char proxy_realm[128];
};

struct struc_server_ports
{
    __int16 tcp;
    __int16 udp;
    __int16 unk;
    __int16 unk_0;
    __int16 unk_1;
    __int16 http;
    __int16 https;
    __int16 tls;
};

enum enum_proto // specified by c2_proto and proxy_proto values
{
    none = 0x0,
    TCP = 0x1,
    HTTP = 0x2,
    HTTPS = 0x3,
    TLS = 0x4,
    UDP = 0x5,
};
```

The C2 protocol of the sample is completely different from the existing understanding of the 3.0 protocol. It will be detailed in the sections that follow.

C2 protocol

As the configuration structure shows, the Worker component supports five C2 protocols: TCP, HTTP, HTTPS, TLS and UDP. While the same customized packet is included in every protocol (e.g. as a raw payload in TCP, as POST data in HTTP, and so on), there are minor differences between the protocols. The following analysis will detail the customized packet format and then the differences between the protocols.

Winnti 4.0 also supports the server-mode function, which accepts incoming packets such as a C2 server for lateral movement. The function was helpful to verify the correctness of the protocol format and encryption handled by the implemented scanner.

Packet format and encryption

The customized packet is separated into the header and payload.

```
struct struc_custom_header
{
    __int16 temp_key_seed;
    __int16 unk_word; // initial value is 2
    __int16 signature; // 0x45DB
    int payload_len;
};
```

The protocol encrypts both the header and payload, except for a `temp_key_seed` value that is provided in the header. The `signature` value is validated after the decryption on the receiver side.

```
struct struc_custom_payload_init
{
    int payload_type; // request:0xEE775BAA/0x4563CEFA/0x5633CBAD, response:0xFACEB007/0x5633CBAD
    int unk_dword; // request:0, response:0xC350/0xC352
    GUID guid;
    char null_bytes[14];
    __int16 seq_num; // starting from 1
    __int16 null_word;
};
```

The initial packet payload contains a payload type, random GUID value, and packet sequence number. The payload type should be one of the following values:

- `0xEE775BAA`
- `0x5633CBAD`
- `0x4563CEFA`
- `0xFACEB007`

Typically, the request payload type will be `0xEE775BAA` and the response will be `0xFACEB007`.

The encryption algorithm is unknown at this time, but it appears to be a stream cipher with no constant values that requires two kinds of keys:

- A dynamically generated key from the `temp_key_seed` value in the header.
- A portion of the SHA1 value of the hard-coded string ‘`host_key`’.

The generation algorithm is implemented in Python:

```
def transform_word(w):
    t = (667 * w) & 0xffff
    t = (t + 4713) & 0xffff
    t = (w * t) & 0xffff
    t = (t + 57) & 0xffff
    t = (w * t) & 0xffff
    t = (t + 1) & 0xffff
    return t

def generate_temp_key(s):
    res = []
    t = transform_word(s)
    res.append(pack('<H', (t * t) & 0xffff))
    t = transform_word(t)
    res.append(pack('<H', (t * t) & 0xffff))
    t = transform_word(t)
    res.append(pack('<H', (t * t) & 0xffff))
    t = transform_word(t)
    res.append(pack('<H', (t * t) & 0xffff))
    t = transform_word(t)
    res.append(pack('<H', (t * t) & 0xffff))
    t = transform_word(t)
    res.append(pack('<H', (t * t) & 0xffff))
    t = transform_word(t)
    res.append(pack('<H', (t * t) & 0xffff))
    t = transform_word(t)
    res.append(pack('<H', (t * t) & 0xffff))
    return ''.join(res)
```

Figure 2: Key generation from a temp_key_seed value.

The unknown encryption/decryption routine can be emulated by IDA Appcall [6]. (The routine will be detailed in the ‘Scanner implementation’ section later in this paper.)

HTTP protocol

The customized packet data itself is the only information transmitted in the TCP/UDP protocols. In the HTTP protocol, this customized packet is sent through a POST request with several HTTP headers, which are depicted in Figure 3.

```
POST /333959650 HTTP/1.1
Host: 127.0.0.1:9999
Connection: keep-alive
Content-Length: 52
Accept: */*
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
50.0.2661.94 Safari/537.36
Content-Type: application/octet-stream
Referer: http://127.0.0.1:9999
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-GB,en;q=0.8
Cookie: 640ABEFB16D2CE36E7E83E1B8BEF31B2500ABEFB

.....x.d..fk.|.I.    Q..G2..91d.....6f1@[.A.+.%..!..h.v
```

Figure 3: HTTP POST request including the customized packet.

It should be noted that the number ‘333959650’ after the ‘POST /’ is randomly generated. The cookie value is made up of five DWORD hex values, which contain information about the customized packet size. The size information is embedded as the XOR key. A Python script was written to parse this data. Figure 4 shows the relevant code from the Python script, which validates the cookie value.

```
dw0 = unpack("<I", binascii.unhexlify(t[:8]))[0]
dw1 = unpack("<I", binascii.unhexlify(t[8:16]))[0]
dw2 = unpack("<I", binascii.unhexlify(t[16:24]))[0]
dw3 = unpack("<I", binascii.unhexlify(t[24:32]))[0]
dw4 = unpack("<I", binascii.unhexlify(t[32:]))[0]
print('dw0={:#x}, dw1={:#x}, dw2={:#x}, dw3={:#x}, dw4={:#x}'.format(dw0, dw1, dw2, dw3, dw4))

if dw0 != dw1 ^ ((dw2 + dw3)&0xffffffff):
    print('dw0 is not correct')
    return
key = dw4 ^ dw0
print('The cookie value validated. dword key = {:#x}'.format(key))
```

Figure 4: Cookie value validation code.

The following shows the output of the Python script, and that the customized packet size is 0x34:

```
$ python validate_cookie.py 640ABEFB16D2CE36E7E83E1B8BEF31B2500ABEFB
dw0=0xfbbe0a64, dw1=0x36ced216, dw2=0x1b3ee8e7, dw3=0xb231ef8b, dw4=0xfbbe0a50
The cookie value validated. dword key = 0x34
```

Prior to the POST request an initial GET request will be made between the client and server. An example of this is shown in Figure 5.

```
GET / HTTP/1.1
Host: 127.0.0.1:9999
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
50.0.2661.94 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-GB,en;q=0.8
Cookie: 420F0DABD80FC8F34050B58A5AB00FCE420F0DAB

HTTP/1.1 200 OK
Cache-Control: public
Content-Type: text/html; charset=utf-8
Set-Cookie: D66EEE1927424A0C7A30387777FC6B9ED66EEE19
Server: Microsoft-IIS/8.5
Content-Length: 2039

<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>DF14F693</title></head>
<body>
'ZX<7;bn@O;X0["s*/_H_i(?x6vFl=#Z30,@wXqNS$-xA)9:t;%}0T.7m3/3<{o9q^O336.^p'A+!ezC)4{,@#4"y5%<7m#r+? xz?!
5E&fd*V<\/-!(n XoD?NV"w"N,K,@BG)iZ=&j2)Rh?:'E[@>_/LW?.8U2=:[a4n0*&6>g!fQoi=Lc 4E?():6$PEC!!U"~?)iE;
3w2.)19H-454Row#Ytv-fm}va>;HP7./>4#"je3R|p?>#.,wXZ0)XXj.!r5j:'2GC/-@RQ}uz5Dl8+' l~>.ZFEA08 V2|P"Q6I6<E?
h4'*':Vw='mY0*$Mah2T:_A)`.DV+d:C;m|Rw?[8h'U$Z<)y33E6!'b(9d84oIB&tY1-. ,2_snB
$geVj)m)D[3SP&'9p;]6+W*j7iAc$63irz%|8w63>9',:<.))>IN)$5YOKIY=ek+QV;)MLc:#5u<-$92`~|z4{wD7cO,EKu;z5,u5-
HDa ^;e*Kj1vJ+mVo:5-kv!enyZM(f^2*Yu>^$mH'>'8m-;A}6!I',3) 1zo1FT+7CD8tcEfM>K%=9EU#;-8E`m'aZ<\*5@_X)\=*/
5m2Z6D1r>caWd~!bhq~OtX$y5VQ"60^A5;w0S~V<$K>G1{&C@"%qm.;=l0?ZNkaHI~&_<.x1MH&klI3Yv$,&wT[,]o9L 1%!pO.
2n,@M:,(K9-2qa/"/\%T#(25@oh^pCZ5$  B0j#)qhZ4gF<^(H>X(0G!Knj{;=(S7':6*V PM1m+.P$-B}a)odx57z]!s12G
+Gw8O19/$ /`<z9h+7Z<)}mSd0O{0R5/AR'ySPbE.8"4:bqGi#%4AVXWjj)hNY(&Q8!z~$i883#=5Hc]r'\XN*6`=%O
\Y8{zYO5'k?'j:Dl@~!q3:k2alfTT5UZ}XOB}g.K,dll1?QF%!-81"6(N09<m@\).9Y^Q(XO&n,?d4'&8x/?"/U}Du6/34Hii%@[B!
p# d3<^OuW 1,8]1v803vetukvN\hBuKl(7{6*%2K):P|Fx_uSXPiw!?7w1:m6hS,^4LB~p*6.,K.AZ:[yG>$\1{DV88: c
2BH"B(4]`I8%)$8I*1/8fora6:*WCG("?t  @g?FO-k&IV1>RU@w< !_JUz5Fd`HEA!/21$6xa 4)6r;t+v 0#6G EB  `6QF
+WQia:!W9f<6C =~?34&gkv(2=)0>J/bT0N#.*c!%:yTR,2p\9*=|ITl"y1Ees#t[9cL)Y4qgU\vV}!1c2o5n"j,,W~;1"6K!
D=;QnG_0$$'ei*o,E]rz#[UyNcZ$$)o5_2MVh>9%2g-QL;:4ok: -%.#&Odyj)W4\K{?> `dM"0]BV37;lwB0}'Zp8&~0$#ZD`d
$6*5f.9\:(49/Jluub:aJ&< &;W0B*r0.;_;;dkX1V?z59/)g&))=8(_EY Nqs,>wW:';J1<sQH)l`i-2U. 4+$)F? 5Gx(x15\"2}E
"/X2 JTg,5;7z&'>.5'KFAO!L)AsFD>-p`3!%17rr.0qk+Az'`R,Jlx,+,PN(V;0&/?!OZefkG%( "kjv !( 'jk&y/)5z<4U|
881*710Xku0DG*';,Y^/Xy*0?;G"3aI)#90+e"-8*xt\9?b>4G4I0+'(&;2gi;d}<m7dYw2um8 FPQ<|n99xy,5#Hl-\%9Ax\m
+9g81d# j44UwFq?o(o_:U@#%]3)V6fGC) 2/*7*P;2N8T/"-V*i-#"O oOHpub?ML0d#:8>A}7Tp:@T_r!^VG|4m/`$!
2$`fu3t/.qv[?-uu+|A_+/BiN;sEsP(G)za;8Emo2hF>B}?~1#+,kU+^%aJ9D8Z"8g3EF<3#sAZ3cv178>;'m'#j<#4%s56?`!
Wwq512T%-CsUq?FCXLsf'^bH;iR]}nT,ubQ %VRM{=*F:0/:9Y<-/pS%uDTNL6&lHC<<LY8</body>
</html>
```

Figure 5: HTTP GET request.

The GET request and response do not contain the customized packet, and subsequently upon decoding the cookie value the size will be shown as zero. The output below shows the decoded cookie value from Figure 5:

```
$ python validate_cookie.py 420F0DABD80FC8F34050B58A5AB00FCE420F0DAB
dw0=0xab0d0f42, dw1=0xf3c80fd8, dw2=0x8ab55040, dw3=0xce0fb05a, dw4=0xab0d0f42
The cookie value validated. dword key = 0x0
$ python validate_cookie.py D66EEE1927424A0C7A30387777FC6B9ED66EEE19
dw0=0x19ee6ed6, dw1=0xc4a4227, dw2=0x7738307a, dw3=0x9e6bfc77, dw4=0x19ee6ed6
The cookie value validated. dword key = 0x0
```

The purpose of the GET request is unknown – it seems redundant, but it could be in place as an initial handshake without revealing the custom packets.

Additionally, it was determined from the configuration that the server-mode code provides two kinds of HTTP server functions:

- A normal HTTP server replying to any request.
- A ‘Reuse’ HTTP server only replying to requests with correct URLs.

The latter HTTP server is implemented by *Windows* HTTP Server APIs and appears to be the same as the passive HTTP listening function of PortReuse reported upon by *ESET* [7]. However, in this implementation the URL is not fixed but rather specified in the configuration block.

```
v2 = HttpCreateHttpHandle(&pReqQueueHandle, 0);
if ( v2 )
  goto LABEL_19;
v2 = HttpAddUrl(pReqQueueHandle, ArgList_1->watch_url, 0i64);// The URL is copied from config
if ( v2 )
  goto LABEL_19;
if ( !CreateIoCompletionPort(pReqQueueHandle, *(HANDLE *)&ArgList_1[1].field_0[8], 0i64, 0) )
  return v2;
v3 = pReqQueueHandle;
*(_DWORD *)&ArgList_1->gap_3C = 0;
*(_QWORD *)ArgList_1[1].field_0 = v3;
*(_DWORD *)&ArgList_1->gap4D[87] = 0;
LODWORD(v4) = beginthreadex(
                0i64,
                0,
                (unsigned int (__stdcall *)(void *))fn_httpapi_receive_and_response,
                ArgList_1,
                0,
                0i64);
```

Figure 6: Reuse HTTP server code.

Additional SSL encryption

The customized packet is also encrypted with SSL when the protocol is HTTPS or TLS. Aside from the added layer of encryption they are identical to HTTP and TCP.

Behaviour after the initial handshake

After the initial handshake, the packet payload contains the command ID and dispatcher ID in a nested structure to control the infected host.

```
struct __declspec(align(4)) struc_custom_payload_next
{
   __int16 messageID;
   __unaligned __declspec(align(1)) int sessionID_src;
   __int16 messageID_0;
   int sessionID_dst; // copied from data sent by C2
   __int16 field_C;
   __int16 field_E;
   __int16 field_10;
   __unaligned __declspec(align(1)) int field_12;
   __unaligned __declspec(align(1)) int field_16;
   __unaligned __declspec(align(1)) int sessionID_dst_0;
   __unaligned __declspec(align(1)) int sessionID_src_0;
   char imm0;
   char unk_byte_from_C2;
   __int16 field_24;
   __unaligned __declspec(align(1)) int field_26;
   __int16 field_2A;
   __int16 field_2C;
   __int16 signature; // 0x45db
   int nested_payload_len;
   struc_nested_payload nested_payload;
};

struct __unaligned __declspec(align(1)) struc_nested_payload // at least 0x14 bytes
{
```

```
    // e.g., cmd_ID=5 & dispatch_ID=1 order to send victim info
    __int16 cmd_ID;
    __int16 dispatch_ID;
    __int16 field_4;
    char field_6;
    char field_7;
    char field_8;
    char field_9;
    char field_A;
    char field_B;
    int ret_value; // -1 or 0 or 1 or 2
    int additional_data_len;
    struc_data_cmd1 additional_data; // flexible size
};
```

However, the Worker component looks to have few built-in remote access trojan (RAT) functionalities other than collecting the host information and executing a PE module plug-in downloaded from the C2. Therefore, I hypothesize either that Winnti 4.0 with the new Worker component must load a plug-in for each desired function like Winnti 3.0, or that the acquired sample was in an early development stage. I haven't acquired any plug-in samples for the Worker yet.

Scanner implementation

I initially had to decide which ports the scanner should focus on. I only had one Winnti 4.0 sample with the new C2 protocol. Therefore, I targeted the following ports based not only on the sample but also on other Winnti malware information reported by *ESET* [7] and *Trend Micro* [8], though neither of these referred to the 4.0 protocols.

- TCP/443
- TCP/80
- UDP/443
- UDP/53

The rough scanning workflow is shown in Figure 7.

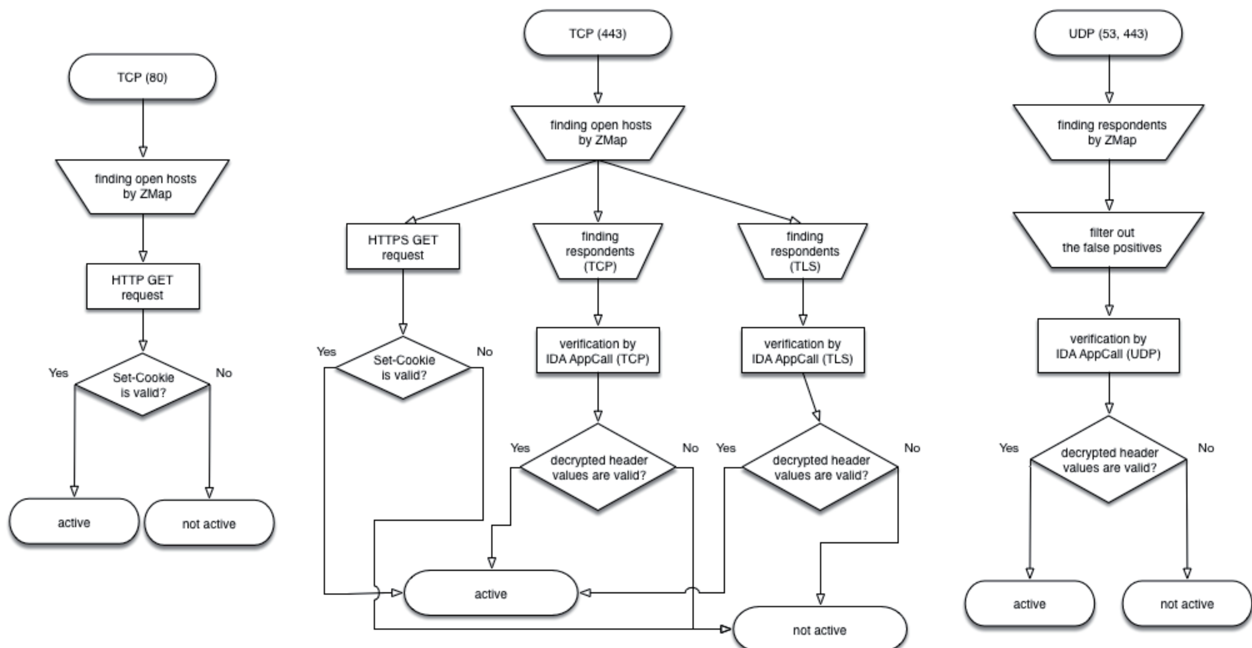

Figure 7: Discovery workflow.

In both TCP-based and UDP protocols, ZMap [9] is first utilized to discover open or respondent IP addresses on the Internet. HTTP/HTTPS GET requests are sent to the hosts with open TCP ports 80/443, then an additional TCP/TLS custom packet transmission with fixed data will be completed for port 443. The TCP/TLS results will be validated by IDA Appcall, based on the decrypted header values (payload size and signature) of the responses. The UDP protocol case can be handled in the same way while the result from ZMap will be filtered out to reduce the noise.

Two scripts were implemented for this discovery. One is a stand-alone Python script validating the HTTP/HTTPS Set-Cookie header values of responses and identifying suspicious respondents to the TCP/TLS customized packets. The

other is an IDAPython script with the Appcall functions encrypting and decrypting the TCP/TLS/UDP packets. Both scripts were tested against the sample where I modified configuration values to enable arbitrary server-mode functions.

The stand-alone script detects not only active hosts returning correct Set-Cookie header values, but also suspicious ones handling different protocols (potential TCP/TLS servers), as below:

```
$ python internet_c2_scan.py -l test.txt -p 80 -d winnti http
[DEBUG] target is "http://172.16.24.127:80"
[DEBUG] validating the Set-Cookie value: 26A3F6E045EC98AD6732D27EFC1C9CCE26A3F6E0
[+] 172.16.24.127,active
[*] 1 scanned in 0:00:00.454756
[*] 1 active servers found

$ python internet_c2_scan.py -l test.txt -p 443 -d winnti tcp
[DEBUG] target is "tcp://172.16.24.127:443"
[+] 172.16.24.127,suspicious,bbef1f956f778eabd17d26b6eb0893658bc896c91a0cdd3fda32c70ba3e6
d9ab3ef9fecee0673537e1e53e6b2e747dce20630e67
[*] 1 scanned in 0:00:00.231333
[*] 1 active servers found
```

The second of the above execution examples should be passed to the IDAPython script as the stand-alone script only checks that the response size is the same and the content is different from the data sent to the IP address.

The IDAPython script validates values in the respondent's packet such as header signature and payload size after the decryption.

```
[*] start
[*] selected: protocol = TCP, port = 443
[DEBUG] created GUID: 0b8212dc-e364-4c18-ac0b-26382beb1387
[DEBUG] client header: unknown word = 0x2, header signature = 0x45db, payload length = 0x2a
[*] client payload: payload type = 0xee775baa, unknown dword = 0x0, GUID = 0b8212dc-e364-4c18-
ac0b-26382beb1387, sequence number = 1
[DEBUG] a request packet sent (52 bytes)
[DEBUG] a response packet received (52 bytes)
[DEBUG] server header: unknown word = 0x2, header signature = 0x45db, payload length = 0x2a
[*] server payload: payload type = 0xfaceb007, unknown dword = 0xc352, GUID = 0b8212dc-e364-4c18-
ac0b-26382beb1387, sequence number = 2
[+] 172.16.24.127: server is active (custom packet validated)
[+] 1 active servers found
[*] done
```

Additionally, we can differentiate between C2s and victim hosts infected with server-mode variants by checking the GUID and sequence number values. As displayed above, the server-mode variants respond with the same GUID (`0b8212dc-e364-4c18-ac0b-26382beb1387`) as the client while incrementing the sequence number. On the other hand, the C2 servers respond with the null GUID and reset the sequence number to 1.

```
[*] start
[*] selected: protocol = TCP, port = 443
[DEBUG] created GUID: 346dfd36-a776-4a2e-9765-a0bcc2524fc9
[DEBUG] client header: unknown word = 0x2, header signature = 0x45db, payload length = 0x2a
[*] client payload: payload type = 0xee775baa, unknown dword = 0x0, GUID = 346dfd36-a776-4a2e-
9765-a0bcc2524fc9, sequence number = 1
[DEBUG] a request packet sent (52 bytes)
[DEBUG] a response packet received (52 bytes)
[DEBUG] server header: unknown word = 0x2, header signature = 0x45db, payload length = 0x2a
[*] server payload: payload type = 0xfaceb007, unknown dword = 0x0, GUID = 00000000-0000-0000-
0000-000000000000, sequence number = 1
[+] 185.161.211.97: server is active (custom packet validated)
[DEBUG] a request packet sent (52 bytes)
[DEBUG] a response packet received (52 bytes)
[DEBUG] server header: unknown word = 0x2, header signature = 0x45db, payload length = 0x2a
[*] server payload: payload type = 0xfaceb007, unknown dword = 0x0, GUID = 00000000-0000-0000-
0000-000000000000, sequence number = 1
[+] 80.82.67.6: server is active (custom packet validated)
[+] 2 active servers found
[*] done
```

The difference in the values can be observed in other protocols like TLS and UDP.

Result

Between December 2019 and May 2022 I discovered 51 Winnti 4.0 C2 servers (39 unique IPs) on the Internet. The percentage of each protocol is shown in Figure 8.

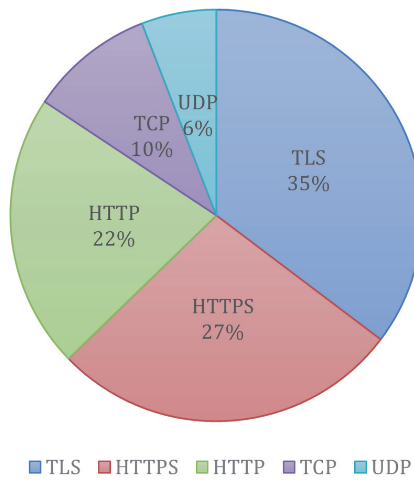

Figure 8: Winnti 4.0 population by protocol.

Based on the GUID and packet sequence number values that I mentioned in the previous section, I conclude that all TLS/TCP/UDP servers were C2s, not infected hosts with the server-mode variants. Unlike these protocols, the HTTP/HTTPS scanners just decode and validate the cookie value. There is no difference in the value between C2s and server-mode infections. However, these were also likely C2s because all servers were hosted by VPS providers and most of the providers overlapped with those hosting TLS/TCP/UDP C2s.

The change in the number of active Winnti 4.0 C2s is shown in Figure 9.

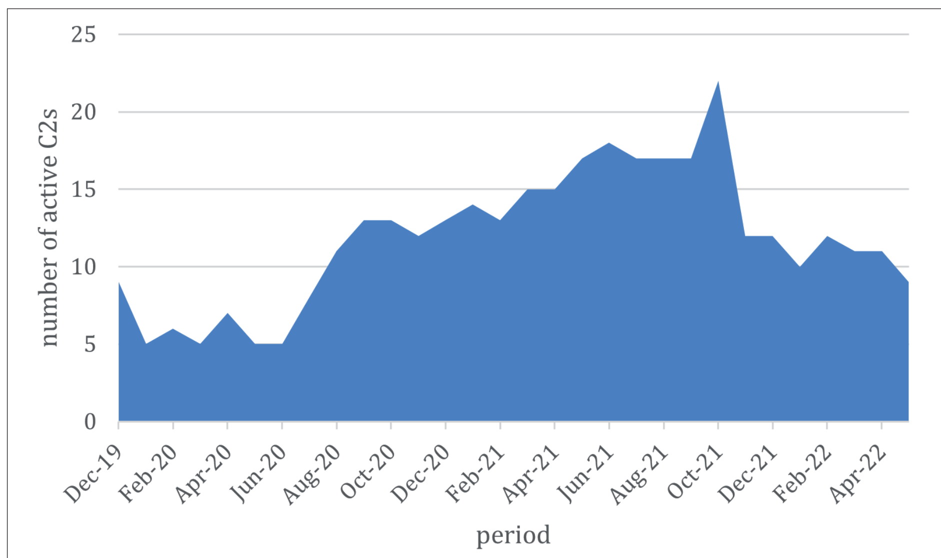

Figure 9: Change in number of active Winnti 4.0 C2s.

I disclosed the discovered Winnti 4.0 C2 information twice (in February 2020 [10] and November 2021 [11]). I suspect that's why the number in November 2021 dropped sharply. However, the new C2 servers are still observed as of the time of writing this paper. To the best of my knowledge, there have been only two public reports – by *Trellix* [12] and *Recorded Future* [13] – on this threat. In order to alert the cybersecurity community, I will continue to track the C2 servers in the future.

ShadowPad

ShadowPad is a modular malware platform privately shared with multiple PRC-linked threat actors since 2015. According to *SentinelOne* [14], ShadowPad is highly likely the successor to PlugX. Due to its prevalence in the cyber espionage field, I was motivated to analyse the C2 protocol to create a scanner.

C2 protocol

ShadowPad supports six C2 protocols: TCP, SSL, HTTP, HTTPS, UDP and DNS. In this research, I focus on TCP/HTTP(S)/UDP protocols, as others like SSL and DNS are not likely utilized by the recent ShadowPad samples.

The format and encoding algorithm are different between TCP and HTTP(S)/UDP.

	TCP	HTTP(S)/UDP
Key size	4	2
Header size	0x14	8
Payload size in the initial handshake packet	Up to 0x3F	HTTP(S): Up to 0x1F, UDP: 0x10

Table 3: Difference in packet format.

The key for the encoding is included in the header. Every integer value in the header is big endian. Randomly sized data will be appended as the payload to the initial handshake packet in both cases.

The immediate values used by the encoding algorithms are different per variant (probably per ShadowPad builder version). I analysed three ShadowPad variants, which I was able to collect in August 2021, as displayed in Table 4. (The SHA256 hash values are included in the Indicators of Compromise section.)

Variant name	C2 protocol	Config size	Attribution	Source
Variant1 (aka ScatterBee [15])	TCP/UDP	0x896	APT41	*Positive Technologies* [16]
Variant2	HTTP(S)	0x85C	Tonto Team	*ESET* [17]
Variant3	HTTP(S)	0x85C	unknown	*VirusTotal* [18]

Table 4: Analysed ShadowPad variants.

The C2 protocol is further detailed below.

TCP protocol

The TCP protocol header format is displayed as follows:

```
struct struc_common_header
{
   int session_key;
   int plugin_and_cmd_id; // plugin_id (0x68) << 16 + cmd_id (0x51)
   int module_code; // 0
   int payload_size_compressed;
   int payload_size_original;
};
```

The header format has been the same since 2015. The `session_key` is randomly generated then used for encoding both the header and the payload. The plugin_id and cmd_id values included in the `plugin_and_cmd_id` field have been updated by variants, some of which are covered in this paper. The values in the initial packet created by Variant1 should be 0x68 (Online plug-in) and 0x51 (check-in). The `module_code` of the initial packet generated by the sender is always 0.

If any payload data exists, it will be compressed with the QuickLZ [19] algorithm. QuickLZ is an older, publicly available compression routine that is not commonly seen. The client generates randomly sized null bytes (up to 0x3F bytes) for the initial packet payload.

Variant1's encoding algorithm for the TCP packet in Python is shown in Figure 10. Based on the protocol analysis experience of Variants 2 and 3, another variant may use a different immediate value from `0x22F4B1BA` for the TCP packet encoding.

```
for s in src:
    key = (key - 0x22F4B1BA) & 0xffffffff
    d = (s ^ (key + (key >> 8) + (key >> 16) + (key >> 24))) & 0xff
    _dst.append(d)
```

Figure 10: TCP packet encoding by Variant1.

After the initial handshake, Variant1 executes the commands of the plug-ins specified by the C2 server. For more details, review the *Dr.Web* white paper [20] explaining the individual command IDs and payload formats. The variant analysed in the paper is older than Variant1 but the formats should be similar.

HTTP(S) and UDP protocols

The header format for the HTTP(S) and UDP protocols is listed below. In HTTP(S), the data is sent through the POST method.

```
struct struc_proto_header
{
   __int16 session_key;
   __int16 type; // 0 in HTTP, req=0x1001/res=(0x2002|0x5005) in UDP
   __int16 session_src_id; // random 2 bytes, generated by both client/server
   __int16 session_dst_id; // req=0, res=client's session_src_id
};
```

The `session_key` has the same role as the TCP `session_key`, though the key size is different. The second field type is always 0 (zero) in the HTTP initial packet, while UDP client and server send 0x1001/0x2002/0x5005. The `session_src_id` field is randomly generated by both client and server. The value sent by the client will be set in the `session_dst_id` field on the server side.

The initial packet payload data are randomly generated based on `QueryPerformanceCounter` and other APIs. The HTTP payload size is also random with a length of up to 31 (0x1F) bytes, while the UDP one is fixed at 16 (0x10) bytes.

Figures 11-13 show each variant's encoding algorithm in Python. The immediate values in the code are different, but the algorithm itself is identical.

```
for s in src:
    tmp1 = (0xCCDD0000 * key) & 0xffffffff
    tmp2 = (0x5A33323 * (key >> 0x10)) & 0xffffffff
    key = (((tmp1 - tmp2) & 0xffffffff) - 0x52B704E3) & 0xffffffff
    d = s ^ (key & 0xff)
    _dst.append(d)
```

Figure 11: UDP packet encoding by Variant1.

```
for s in src:
    tmp1 = (0xAD5E0000 * key) & 0xffffffff
    tmp2 = (0x1C1A52A2 * (key >> 0x10)) & 0xffffffff
    key = (((tmp1 - tmp2) & 0xffffffff) - 0x43B69C62) & 0xffffffff
    d = s ^ (key & 0xff)
    _dst.append(d)
```

Figure 12: HTTP(S) packet encoding by Variant2.

```
for s in src:
    tmp1 = (0x8D7B0000 * key) & 0xffffffff
    tmp2 = (0x633D7285 * (key >> 0x10)) & 0xffffffff
    key = (((tmp1 - tmp2) & 0xffffffff) - 0x7950BEA0) & 0xffffffff
    d = s ^ (key & 0xff)
    _dst.append(d)
```

Figure 13: HTTP(S) packet encoding by Variant3.

After the initial handshake, the payload will contain the same data structure as the TCP packet (struc_common_header and its QuickLZ-compressed payload) explained in the previous section, while the type field in struc_proto_header will be incremented.

Scanner implementation

I decided the following target protocols/ports based on the configurations extracted from recent ShadowPad samples. As explained earlier, the scanner per variant had to be implemented due to the difference in immediate values used in the encoding.

Scanning start period	Target protocol/port/variant
September 2021	HTTP/443 (Variant2 & Variant3)
October 2021	TCP/443 & UDP/53 (Variant1)
June 2022	UDP/443 (Variant1), HTTP/80 (Variant3)

The flow chart in Figure 14 shows how the ShadowPad C2 servers are detected by the scanners.

Like Winnti 4.0 C2 scanning, first the list of hosts open at targeted ports is created by ZMap. Then the scanner sends the ShadowPad-formatted packets to all IP addresses on the list. Next, the scanner checks that the response packet size is at least greater than the header size and the `session_key` is different from the sending one to exclude honeypots. If the size

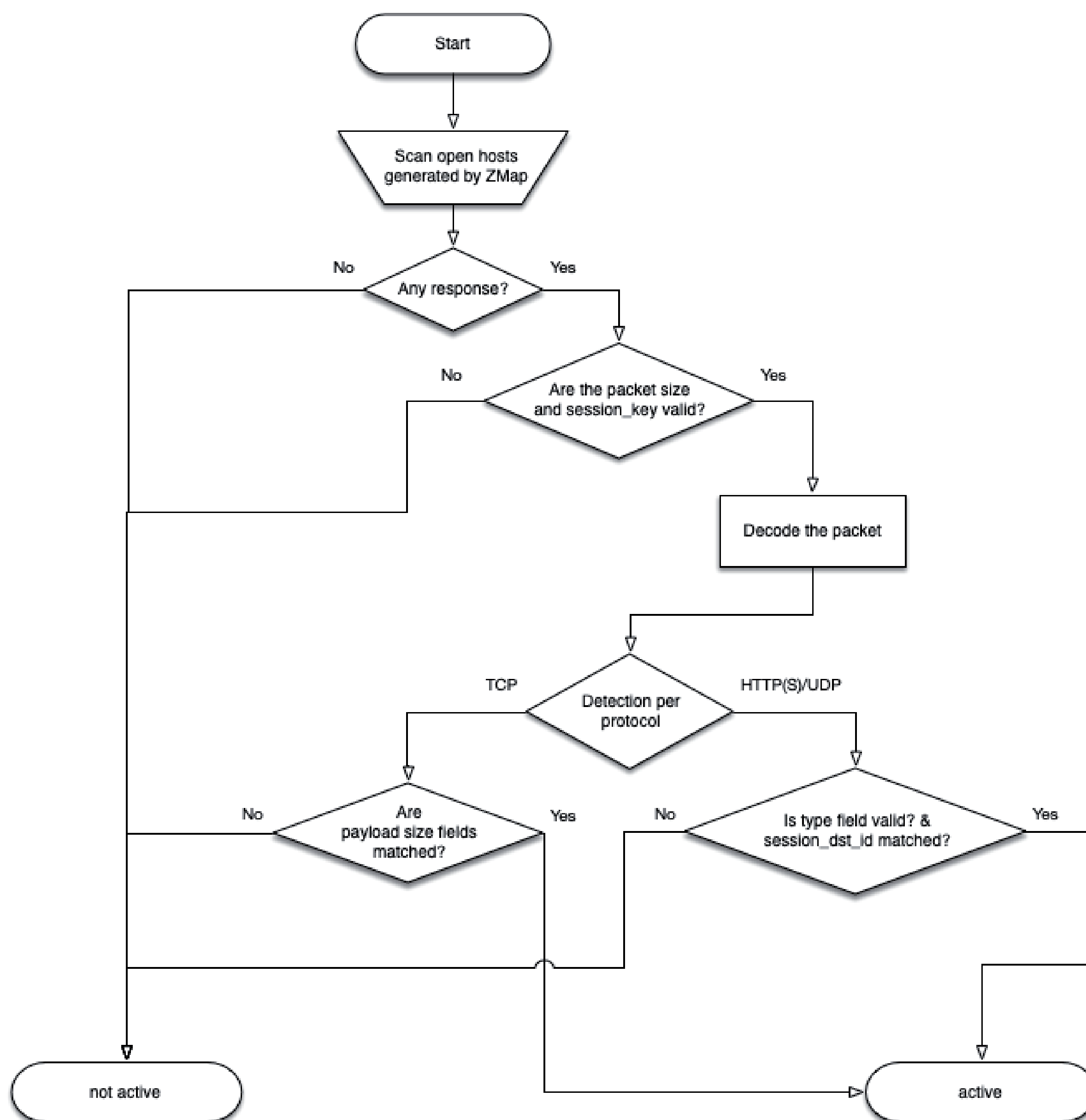

Figure 14: ShadowPad C2 detection flow.

and key look to be valid, the scanner decodes the response packet. In TCP protocol, the scanner validates the payload size fields (`payload_size_compressed` and `payload_size_original`). In HTTP(S) and UDP protocols, the code verifies if the type field value is correct and the response's `session_dst_id` is matched with the `session_src_id` created by the scanner.

The following output log shows that eight Variant1 TCP servers were discovered by scanning the list of TCP/443 open hosts generated by ZMap. The `command_id` 0x53 from the C2s is a request to send system information of the infected host.

```
2022/06/xx xx:00:02,log file opened: scan_results/sp_scan_auto_202206xx_xxxxxx.csv
2022/06/xx xx:00:05,malware options: family = ShadowPad; targeted protocol = tcp (version = Variant1)
2022/06/xx xx:00:09,ShadowPad specific options: version = Variant1; key size = 4; key endian = big; header size = 0x14; Online plugin ID = 0x68; CMD ID = 0x51; module code = 0x0
2022/06/xx xx:00:16,51576779 open hosts read from corpus/2022-xx-xx_zmap22000ppsVPN_tcp_443.saddr
2022/06/xx xx:43:46,45.137.10.3,active,compressed payload size matched (plugin_id=0x68, command_id=0x53, payload=None)
2022/06/xx xx:40:28,45.32.248.92,active,compressed payload size matched (plugin_id=0x68, command_id=0x53, payload=None)
```

```
..[SKIPPED]..
2022/06/xx xx:01:05,43.129.188.223,active,compressed payload size matched (plugin_id=0x68,
command_id=0x53, payload=None)
2022/06/xx xx:48:35,51576779 scanned in 1 day, 17:48:32.497550
2022/06/xx xx:48:35,8 suspicious/active servers found (DB new=4 update=4)
```

In order to detect the Variant2/Variant3 C2 servers I just use the HTTP protocol scanner, not the HTTPS one, because the ShadowPad C2s can accept multiple protocol requests at a single port. I noticed the unique feature by extracting the C2 server configurations from the sample (SHA256: d011130defd8b988ab78043b30a9f7e0cada5751064b3975a19f4de92d2c0025).

```
[*] config size = 0x85c
..
[+] C2 Entry 0 (offset 0xbc): 'HTTPS://wwa1we.wbew.amazon-corp.wikaba.com:443'
[+] C2 Entry 1 (offset 0xed): 'HTTP://wwa1we.wbew.amazon-corp.wikaba.com:443'
..
```

The hostnames and ports in the entries matched exactly but the protocols were different. In fact, I was able to verify that another active ShadowPad C2 can accept both protocols at the same port.

```
$ ./c2fs.py -d -l corpus/query.txt -p 443 -f sp http Variant2
..
[*] malware options: family = ShadowPad; targeted protocol = http (version = Variant2)
[*] ShadowPad specific options: version = Variant2; key size = 2; key endian = big; header size =
0x8; header type = 0x0; client session ID = 53978
[D] POST: http://137.220.185.203:443/ (proxy={}, stream=True, timeout=30)
[+] 137.220.185.203,active,client session ID matched (type=0x0)
..

$ ./c2fs.py -d -l corpus/query.txt -p 443 -f sp https Variant2
..
[*] malware options: family = ShadowPad; targeted protocol = https (version = Variant2)
[*] ShadowPad specific options: version = Variant2; key size = 2; key endian = big; header size =
0x8; header type = 0x0; client session ID = 52256
[D] POST: https://137.220.185.203:443/ (proxy={}, stream=True, timeout=30)
[+] 137.220.185.203,active,client session ID matched (type=0x0)
..
```

The same behaviour may be seen in other protocol combinations such as TCP/SSL and UDP/DNS. However, it’s impossible to test because I have not yet obtained any samples of the variants with multiple C2 protocol plug-ins.

Result

Between September 2021 and June 2022 I identified 72 ShadowPad C2 servers (67 unique IPs) on the Internet. The percentage of each variant is shown in Figure 15. Through the tracking period, Variant1 had become more active.

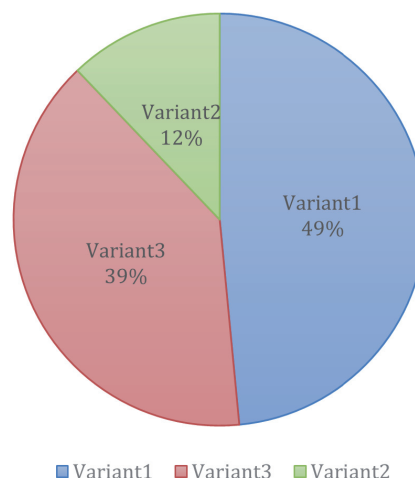

Figure 15: ShadowPad population by variant.

The change in the number of active ShadowPad C2s is shown in Figure 16.

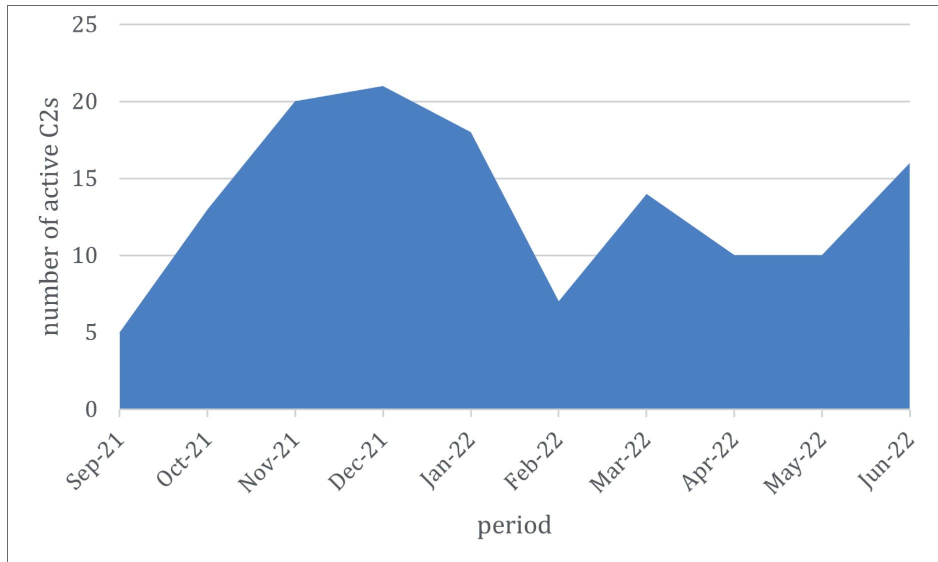

Figure 16: Change in number of active ShadowPad C2s.

Compared with 2021, the active C2s in 2022 have been on a declining trend, though the sharp drop in February 2022 was due to a system issue. The scanner may have missed a new variant lately as ShadowPad changes the immediate values used in the packet encoding per variant. I will continue to improve the scanner as I obtain new variant samples.

As ShadowPad is more prevalent than Winnti 4.0, I was able to identify three samples communicating with the C2 IP addresses on *VirusTotal*. The sample information is listed in Table 5.

Sample malware family	C2 IP address	C2 protocol/port used by sample	Sample submission date	C2 first-seen date by scanner	C2 last-seen date by scanner
Spyder [21]	156.240.104.149	TLS/443	2021/10/26	2021/10/16	2021/10/16
ReverseWindow [22]	43.129.188.223	TCP/10333	2022/02/27	2021/10/17	2022/06/14
ShadowPad	213.59.118.124	UDP/443	2022/03/20	2022/03/06	2022/06/13

Table 5: Samples communicating with the ShadowPad C2 IPs.

Spyder and ReverseWindow are APT malware utilized by PRC-linked cyber espionage threat actors (respectively APT41 and LuoYu). All C2s were discovered by the TCP/443 Variant1 scanner, but the samples communicated with a different protocol or port. With the exception of the Spyder sample case, the C2s had accepted multiple protocols/ports at that time. The scanning system caught the C2s prior to the sample submissions.

Additionally, last year the discovery of the use of a discovered C2 IP (107.155.50.198) triggered an incident response. The advanced and sophisticated attack had bypassed many methods of detection but was ultimately alerted upon simply because of the pre-identified C2 IP.

Figure 17: Alert based on the ShadowPad C2.

NOTES FOR INTERNET-WIDE C2 SCANNING

How to get input data

As explained in the 'Scanner Implementation' sections, the scanner system needs lists of the IP addresses open at targeted ports (in short, port scan data) as the input because it is not realistic to scan full ranges of IPs on the Internet with the

Python-based system. The lists can be obtained by purchasing from commercial services such as *Shodan* [23] and *CenSys* [24], but I generate them myself using *ZMap* for the following reasons:

- For UDP-based protocols, we must scan hosts with the customized protocol formats. For instance, ShadowPad C2s binding to UDP/53 will not answer to the DNS query packets. Commercial services normally don't provide an option to scan with a customized packet.
- As shown in Table 6, commercial services don't scan minor ports actively. For example, *Shodan* has published the scanning target ports [25] on its website. If threat actors avoid using these ports, we will not be able to get the effective input data from *Shodan*.

	ZMap	Shodan	CenSys
TCP/10333	4,940,037	4	1,306
TCP/55555	3,199,856	86	486,497

Table 6: The number of IPs open at minor ports (November 2021).

Anonymization

Today, a lot of organizations – not only criminals but also companies/institutes – scan the Internet for various purposes. Even if the purpose is legitimate, the scanning operations are sometimes forcibly terminated by ISPs and VPS providers due to abuse reports by external network administrators. In order to sustain the C2 scanning research, the source address should be anonymized.

There are two methods for the anonymization: using Tor or commercial VPN services. A comparison between the two methods is shown in Table 7.

	Tor	Commercial VPN service
Supported protocols	TCP	TCP/UDP
Cost	Free	Non-free
Risk of being blocked	High	Low

Table 7: Comparison between anonymization methods.

Tor is free and open-source software to enable anonymous communication in TCP-based protocols. Tor has a cost advantage, but it also has a risk of being blocked easily because the Tor exit relay list is published [26]. On the other hand, commercial VPN services like *Mullvad* [27] have a cost attached, but support UDP-based protocols. Additionally, it's hard for threat actors to block all commercial VPN servers. Therefore, I utilize one of the commercial VPN services for the anonymization.

We must use the '-X' option in *ZMap* for non-Ethernet interfaces like VPN. However, *ZMap* contains a bug causing a segmentation fault when using this option. I have reported this bug and provided a patch within a *ZMap* issue [28]. While the bug has not yet been patched, I highly recommend performing your own patch by using the code explained on that issue page.

CONCLUSION

By emulating the Winnti 4.0 and ShadowPad C2 protocols then scanning the C2 servers on the Internet, I've discovered over 120 C2 servers. The IOCs have been published on *GitHub* [29] with discovered date ranges which are more helpful than just IP address information since the C2s are typically found on hosted servers.

65% of these IOCs have zero detections on *VirusTotal*. Approximately 10 C2s have always been active in both malware families. I see little possibility of false positives because the C2 protocol formats and encoding algorithms are fairly unique. In fact, I've never received false positive feedback regarding the malware families since the IOCs came into use for endpoint detection while I could identify both internal and external infection cases.

Scanning APT malware C2s on the Internet is sometimes as hard as finding a needle in a haystack. However, once the C2 scanning works, it can become a game changer as one of the most proactive threat detection approaches.

ACKNOWLEDGEMENT

I appreciate Tadashi Kobayashi's insight and advice. Kobayashi provided tactics for the scalable scanner implementation and troubleshooting.

I also appreciate Leon Chang's expertise and advice regarding ShadowPad. Chang shared his knowledge to get a smart, big picture of the variants.

INDICATORS OF COMPROMISE (IOC)

Indicator	Type	Context
0a3279bb86ff0de24c2a4b646f24ffa196ee639cc23c64a044e20f50b93bda21	SHA256	Winnti 4.0 dat file
03b7b511716c074e9f6ef37318638337fd7449897be999505d4a3219572829b4	SHA256	ShadowPad Variant1
aef610b66b9efd1fa916a38f8ffea8b988c20c5deebf4db83b6be63f7ada2cc0	SHA256	ShadowPad Variant2
d011130defd8b988ab78043b30a9f7e0cada5751064b3975a19f4de92d2c0025	SHA256	ShadowPad Variant3
1ded9878f8680e1d91354cbb5ad8a6960efd6ddca2da157eb4c1ef0f0430fd5f	SHA256	Spyder communicating with the ShadowPad C2 (156.240.104.149)
536def339fefa0c259cf34f809393322cdece06fc4f2b37f06136375b073dff3	SHA256	ReverseWindow communicating with the ShadowPad C2 (43.129.188.223)
9447b75af497e5a7f99f1ded1c1d87c53b5b59fce224a325932ad55eef9e0e4a	SHA256	ShadowPad Variant1 communicating with the ShadowPad C2 (213.59.118.124)

REFERENCES

[1] FireEye. Double Dragon: APT41, a dual espionage and cyber crime operation. https://content.fireeye.com/apt-41/rpt-apt41/.

[2] Kaspersky Lab Global Research and Analysis Team. “Winnti” More than just a game. April 2013. https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/20134508/winnti-more-than-just-a-game-130410.pdf.

[3] Novetta. WINNTI ANALYSIS. https://www.novetta.com/wp-content/uploads/2015/04/novetta_winntianalysis.pdf.

[4] Macnica Networks. APTマルウェアに見る不易流行. Japan Security Analyst Conference 2018. https://www.jpcert.or.jp/present/2018/JSAC2018_09_yanagishita-takeuchi.pdf.

[5] Threat Analysis Unit. CB TAU Threat Intelligence Notification: Winnti Malware 4.0. VMware Security Blog. September 2019. https://blogs.vmware.com/security/2019/09/cb-tau-threat-intelligence-notification-winnti-malware-4-0.html.

[6] Hex-Rays. IDA Pro 5.6 Appcall user guide. 2010. https://www.hex-rays.com/wp-content/uploads/2019/12/debugging_appcall.pdf.

[7] M.Léveillé, M.-E.; Tartare, M. Connecting the dots: Exposing the arsenal and methods of the Winnti Group. We Live Security. October 2019. https://www.welivesecurity.com/2019/10/14/connecting-dots-exposing-arsenal-methods-winnti/.

[8] Trend Micro. Winnti Abuses GitHub for C&C Communications. March 2017. https://www.trendmicro.com/en_us/research/17/c/winnti-abuses-github.html.

[9] Zmap. https://github.com/zmap/zmap.

[10] Threat Analysis Unit. Threat Analysis: Active C2 Discovery Using Protocol Emulation Part2 (Winnti 4.0). VMware Security Blog. February 2020. https://blogs.vmware.com/security/2020/02/threat-analysis-active-c2-discovery-using-protocol-emulation-part2-winnti-4-0.html.

[11] Haruyama, T. Monitoring Winnti 4.0 C2 Servers for Two Years. VMware Security Blog. November 2021. https://blogs.vmware.com/security/2021/11/monitoring-winnti-4-0-c2-servers-for-two-years.html.

[12] Beek, C. Operation ‘Harvest’: A Deep Dive into a Long-term Campaign. Trellix. September 2021. https://www.trellix.com/en-us/about/newsroom/stories/threat-labs/operation-harvest-a-deep-dive-into-a-long-term-campaign.html.

[13] Insikt Group. China-Linked Group TAG-28 Targets India’s “The Times Group” and UIDAI (Aadhaar) Government Agency With Winnti Malware. Recorded Future. September 2021. https://go.recordedfuture.com/hubfs/reports/cta-2021-0921.pdf.

[14] Hsieh, Y-J. ShadowPad | A Masterpiece of Privately Sold Malware in Chinese Espionage. SentinelOne. August 2021. https://www.sentinelone.com/labs/shadowpad-a-masterpiece-of-privately-sold-malware-in-chinese-espionage/.

[15] Prescott, A. Chasing Shadows: A deep dive into the latest obfuscation methods being used by ShadowPad. PwC. December 2021. https://www.pwc.co.uk/issues/cyber-security-services/research/chasing-shadows.html.

[16] Positive Technologies. Higaisa or Winnti? APT41 backdoors, old and new. January 2021. https://www.ptsecurity.com/ww-en/analytics/pt-esc-threat-intelligence/higaisa-or-winnti-apt-41-backdoors-old-and-new/.

[17] Faou, M.; Tartare, M.; Dupuy, T. Exchange servers under siege from at least 10 APT groups. We Live Security. March 2021. https://www.welivesecurity.com/2021/03/10/exchange-servers-under-siege-10-apt-groups/.

[18] VirusTotal. https://www.virustotal.com/gui/file/d011130defd8b988ab78043b30a9f7e0cada5751064b3975a19f4de92d2c0025.

[19] QuickLZ. http://www.quicklz.com/.

[20] Dr.Web. Study of the ShadowPad APT backdoor and its relation to PlugX. October 2020. https://st.drweb.com/static/new-www/news/2020/october/Study_of_the_ShadowPad_APT_backdoor_and_its_relation_to_PlugX_en.pdf.

[21] Dr.Web. Study of the Spyder modular backdoor for targeted attacks. March 2021. https://st.drweb.com/static/new-www/news/2021/march/BackDoor.Spyder.1_en.pdf.

[22] Leon; Shui. “LuoYu” The eavesdropper sneaking in multiple platforms. TeamT5. https://jsac.jpcert.or.jp/archive/2021/pdf/JSAC2021_301_shui-leon_en.pdf.

[23] Shodan. https://www.shodan.io/.

[24] CenSys. https://censys.io/.

[25] https://api.shodan.io/shodan/ports.

[26] Tor. I want to ban the Tor network from my service. https://support.torproject.org/abuse/i-want-to-ban-tor/.

[27] Mullvad. https://mullvad.net/en/.

[28] Haruyama, T. Segmentation fault when sending IP layer packets #580. https://github.com/zmap/zmap/issues/580.

[29] https://github.com/carbonblack/active_c2_ioc_public.