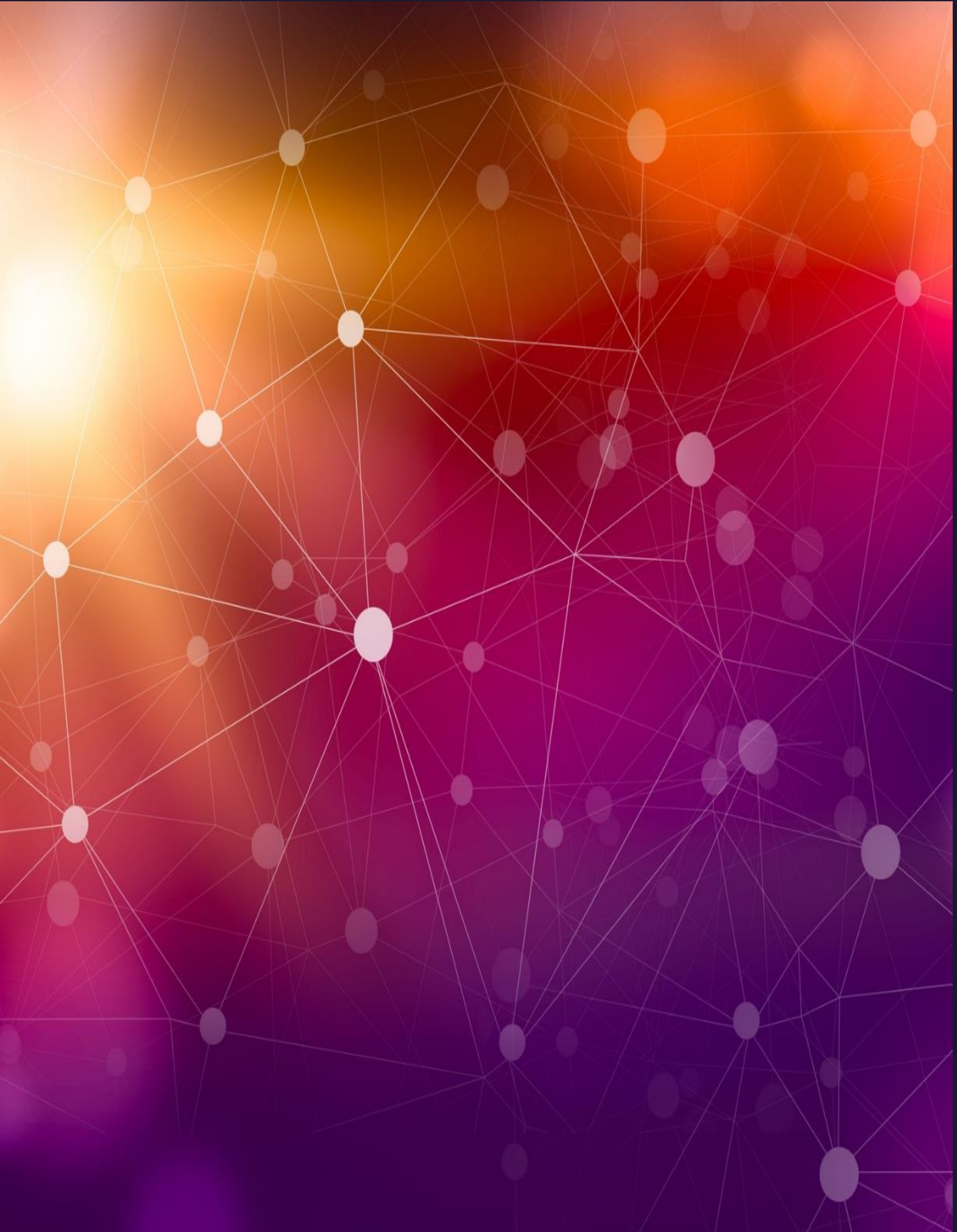


Tracking the entire iceberg

- long-term APT malware C2 protocol emulation and scanning

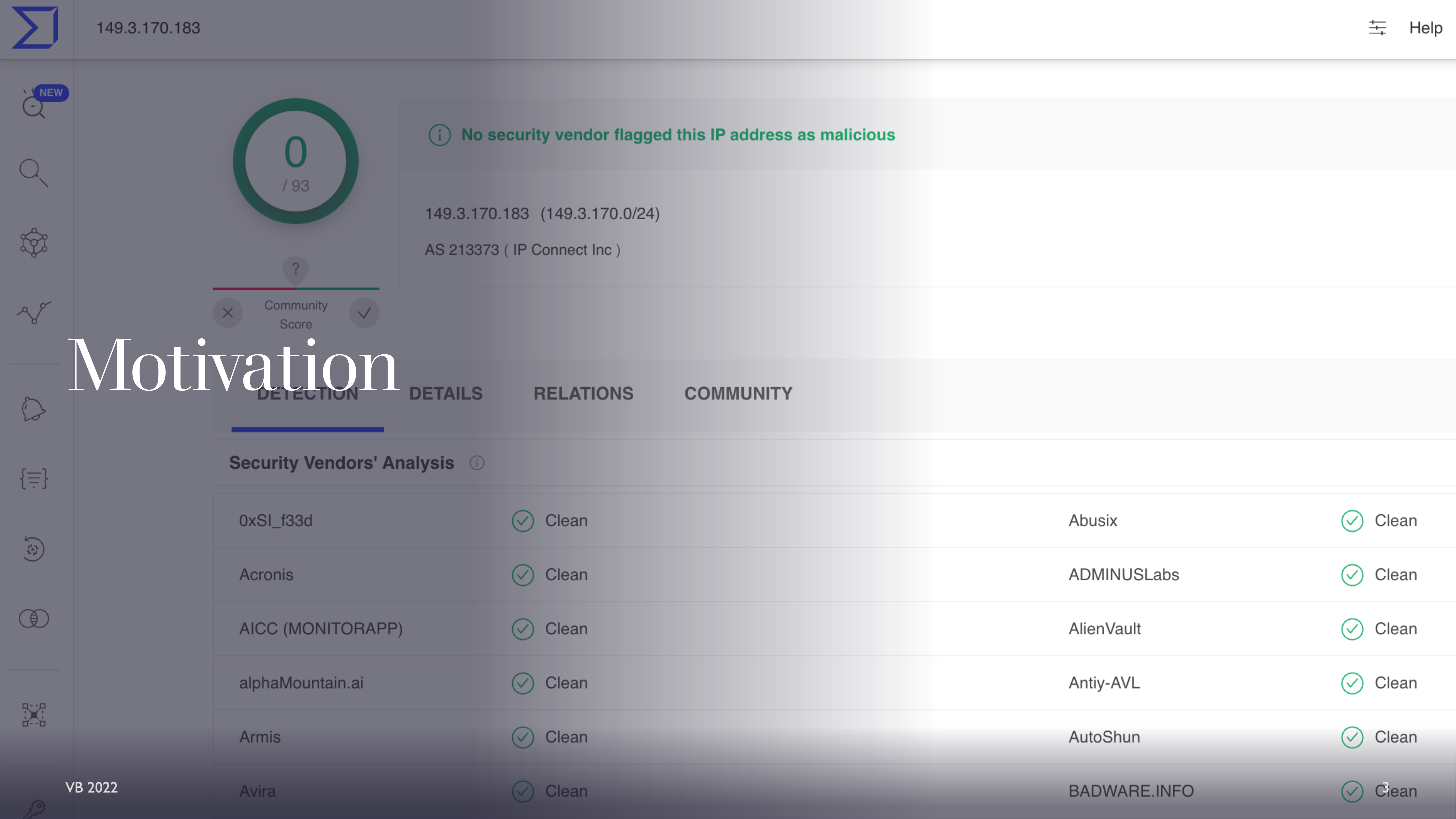
Takahiro Haruyama

VMware Threat Analysis Unit

Who am I?

- Takahiro Haruyama (@cci_forensics)
  - Senior Threat Researcher at VMware Carbon Black TAU
- Past Research
  - Anti-Forensics (e.g., firmware acquisition MitM attack)
  - RE (e.g., defeating compiler-level obfuscations)
  - Malware Analysis (e.g., Cobalt Strike C2 scanning)

Motivation


149.3.170.183
Help
NEW
0
/ 93
No security vendor flagged this IP address as malicious
149.3.170.183 (149.3.170.0/24)
AS 213373 (IP Connect Inc)
?
Community Score
DETECTION
DETAILS
RELATIONS
COMMUNITY
Security Vendors' Analysis
0xSI_f33d Clean
Abusix Clean
Acronis Clean
ADMINUSLabs Clean
AICC (MONITORAPP) Clean
AlienVault Clean
alphaMountain.ai Clean
Antiy-AVL Clean
Armis Clean
AutoShun Clean
Avira Clean
BADWARE.INFO Clean

Overview

- Target Summary
- Winnti 4.0
- ShadowPad
- Notes for Internet-wide C2 Scanning
- Wrap-up

Target Summary

	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

Winnti 4.0

Winnti Malware

- Seen in many large scale attacks and has been attributed to APT41 at the least
- 2013: First reported by Kaspersky (version 1.0-2.0)
- 2015: Novetta analyzed the start-up sequence and C2 protocol of version 3.0
  - Winnti 3.0 Components
    - Dropper -> Service -> Worker (payload)

Winnti Malware 4.0

- 2016: The version 3.0 start-up sequence changed
  - Macnica Networks first described the new variants at JSAC 2018
  - I refer to the variants as version 4.0

	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

Winnti Malware 4.0 (Cont.)

```
struct struc_work_config {
  char campaignID[64];
  char MAC_addr[6];
  int c2_proto; // enum_proto
...
}
enum enum_proto {
  none = 0x0,
  TCP = 0x1,
  HTTP = 0x2,
  HTTPS = 0x3,
  TLS = 0x4,
  UDP = 0x5,
};
```

- 2018: A new Worker component identified
  - < 50% similarity with the 3.0 Workers
    - The C2 protocol was completely different

C2 Protocol

- 5 protocols supported
  - TCP (TLS), HTTP(S), and UDP
- The same customized packet is handled in every protocol
- Server-mode accepting incoming packets
  - Behave like a C2 server
  - Helpful to verify the correctness of the protocol format and encryption

Packet Format

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

```
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;
};
```

Encryption

```
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)
```

- The algorithm is unknown ☹
  - Stream cipher with no constant values?
  - I emulate it using IDA AppCall
- Two kinds of keys
  - Dynamically-generated key from the temp_key_seed
  - portion of the SHA1 value of a hard-coded string "host_key"

HTTP Protocol

- The customized packet is sent through a POST request with several HTTP headers
  - The Cookie value contains its packet size

```
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
```

Customized packet size

Customized packet

HTTP: Size Calculation from Cookie Value

- Made up of 5 DWORD hex values
  - dw0|dw1|dw2|dw3|dw4 in little endian
  - dw0 = dw1 ^ (dw2 + dw3)
  - dw4 ^ dw0 = customized packet size

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

HTTP: Dummy Data in GET Request

```
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 Gec
50.0.2661.94 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-GB,en;q=0.8
Cookie: 420F0DABD80FC8F34050B58A5AB00FCE420F0DAB    Size = 0

HTTP/1.1 200 OK
Cache-Control: public
Content-Type: text/html; charset=utf-8
Set-Cookie: D66EEE1927424A0C7A30387777FC6B9ED66EEE19    Size = 0
Server: Microsoft-IIS/8.5
Content-Length: 2039

<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>DF14F693</title></head>
<body>
'ZX<7;bn@0;X0["s*/_H_i(?x6vFl=#Z30,@wXqNS$-xA)9:t;%}0T.7m3/3<{o9q^0336.^p'A+!ezC)4{
5E&fd*V<\/-!(n_XoD?NV"w"N.K.@BG)iZ=&i2)Rh?:'E[@>_/LW?.8U2=;[a4n0*&6>g!f0oi=Lc_4E?()
```

- Prior to the POST request, an initial GET request will be made
- But the request/response contain no customized packet
  - We can verify it by decoding the size info

Behavior After the Initial Handshake

- Few built-in RAT functions like 3.0
  - Most commands are related to plugin management

```
struct struc_custom_payload_next
{
  __int16 messageID;
...
__int16 signature; // 0x45db
  int nested_payload_len;
  struc_nested_payload nested_payload;
};
```

```
struct 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;
...
int additional_data_len;
  struc_data_cmd1 additional_data; // flexible size
};
```

Scanner Implementation

ZMap

- Internet-wide port scan
  - TCP 443 & 80
  - UDP 443 & 53 (customized packet required)

Stand-alone Python Script

- HTTP(S): Decode and Validate Cookie value
- Others: Get suspicious responses with the same size and different key

IDAPython AppCall

- Decrypt response's customized packet
- Validate signature and payload size in the header

How to Differentiate Server-mode Infections and C2 Servers

- Check GUID (guid) and packet sequence number (seq_num) in the decrypted payload

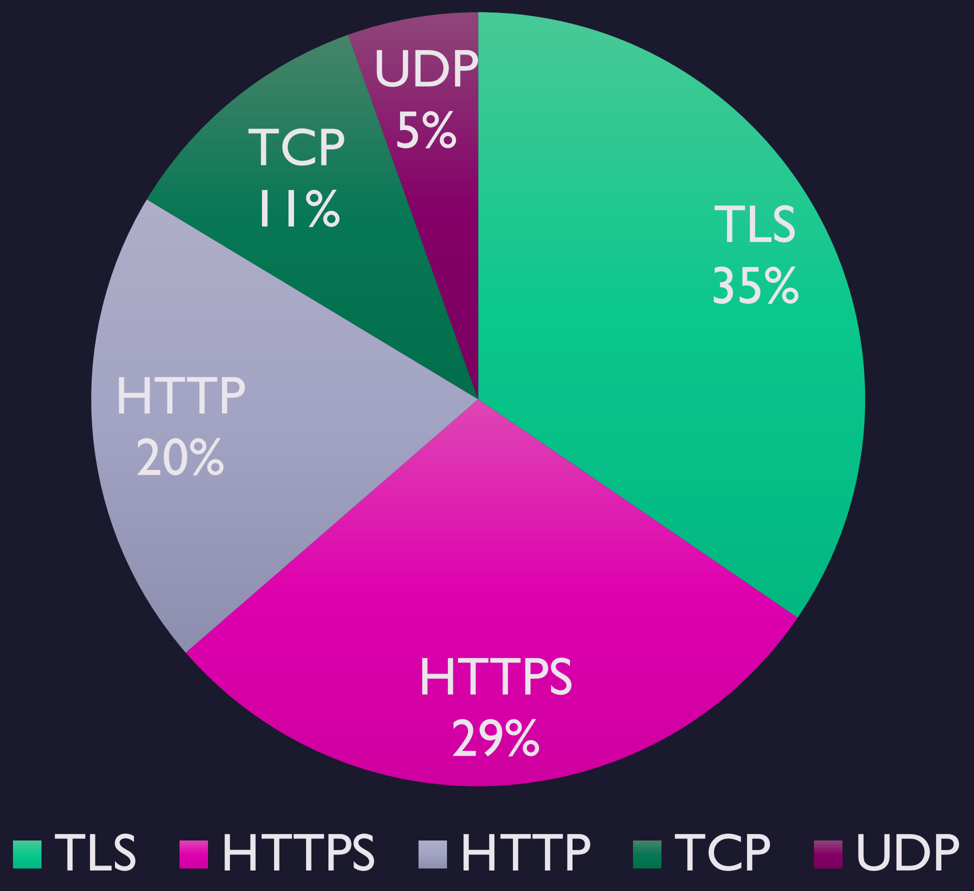
Server-mode: the same GUID as client, sequence number incremented

[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

C2: null GUID, sequence number reset

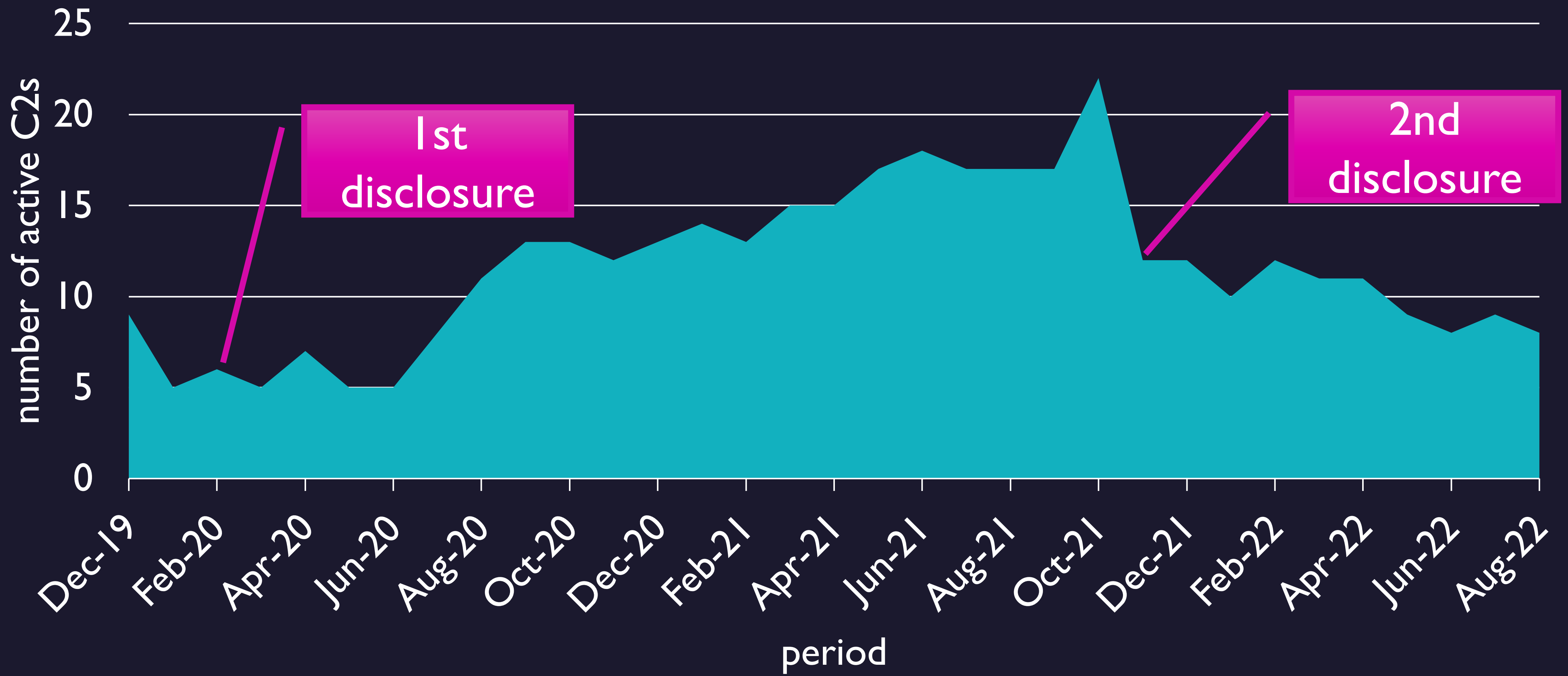
[DEBUG] server header: unknown word = 0x2, header signature = 0x45db, payload length = 0x2a
[*] server payload: payload type = 0xfaceb007, unknown dword = 0x0, GUID = 00000000-0000-0000-0000-000000000000, sequence number = 1

Result: Population by Protocol



- 2019/12 - 2022/08
- 55 servers, 43 unique IPs
- All were likely C2s
  - TCP/TLS/UDP
    - Validated by payload values
  - HTTP(S)
    - Hosted by VPS providers which are overlapped with TLS/TCP/UDP C2s

Result: Change in Number of Active C2s



Public Reports Related to Winnti 4.0 C2s

- Only 2 external reports published in 2021/09
  - Trellix
    - 185.161.211.97 discovered by an incident response
    - Our scanning system identified it since 2019/12
  - Recorded Future
    - 185.161.209.87 and 86.107.197.182 discovered by a similar approach?
    - Our system detected at least in 2021/01

ShadowPad

ShadowPad Malware

- Modular malware platform privately-shared with multiple PRC-linked threat actors since 2015
  - "The successor to PlugX" (SentinelOne)
- 6 C2 protocols supported
  - TCP, SSL, HTTP, HTTPS, UDP, and DNS
- I focus on TCP/HTTP(S)/UDP
  - SSL and DNS are not likely utilized by the recent samples

C2 Protocol

- The format and encoding algorithm are different between TCP and HTTP(S)/UDP
- Randomly-sized data will be appended as the payload

	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

C2 Protocol (Cont.)

- The immediate values used by the encoding algorithms are different per variant
  - Probably per ShadowPad builder version?
- I analyzed three variants collected in August 2021

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

TCP Protocol

- The header format has been the same since 2015
- The payload is compressed with [QuickLZ]
  - For the initial packet, randomly-sized null bytes generated

```
struct struc_common_header
{
  int session_key;
  int plugin_and_cmd_id; // plugin_id (0x68) << 16 + cmd_id (0x51) by Variant1
  int module_code; // 0
  int payload_size_compressed; // QuickLZ
  int payload_size_original;
};
```

TCP Protocol (Cont.)

- Only Variant1 had the TCP plugin for C2 protocol
  - Another variant may use different immediate value for the encoding
- After the initial handshake, ShadowPad executes the commands of the plugins specified by the C2 server
  - For the individual command IDs and payload formats, refer to Dr.WEB white paper

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

HTTP(S) and UDP Protocols

- The header/payload are sent
  - as raw data in UDP
  - through a POST method in HTTP(S)
- The initial packet payload data are randomly generated

```
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
};
```

HTTP(S) and UDP Protocols (Cont.)

- The immediate values in the packet decoding code are different per variant, but the algorithm 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)
```

UDP packet encoding by Variant1

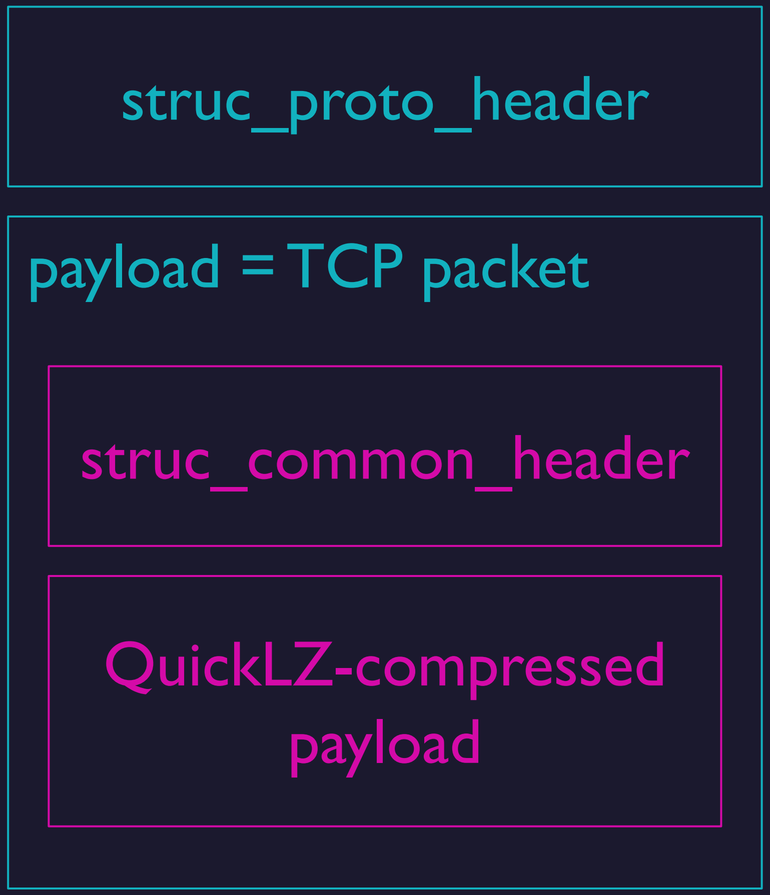
```
for s in src:
    tmp1 = (0xAD5E0000 * key) & 0xffffffff
    tmp2 = (0x1C1A52A2 * (key >> 0x10)) & 0xffffffff
    key = (((tmp1 - tmp2) & 0xffffffff) - 0x43B69C62) & 0xffffffff
    d = s ^ (key & 0xff)
    _dst.append(d)
```

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)
```

HTTP(S) packet encoding by Variant3

HTTP(S) and UDP Protocols (Cont.)



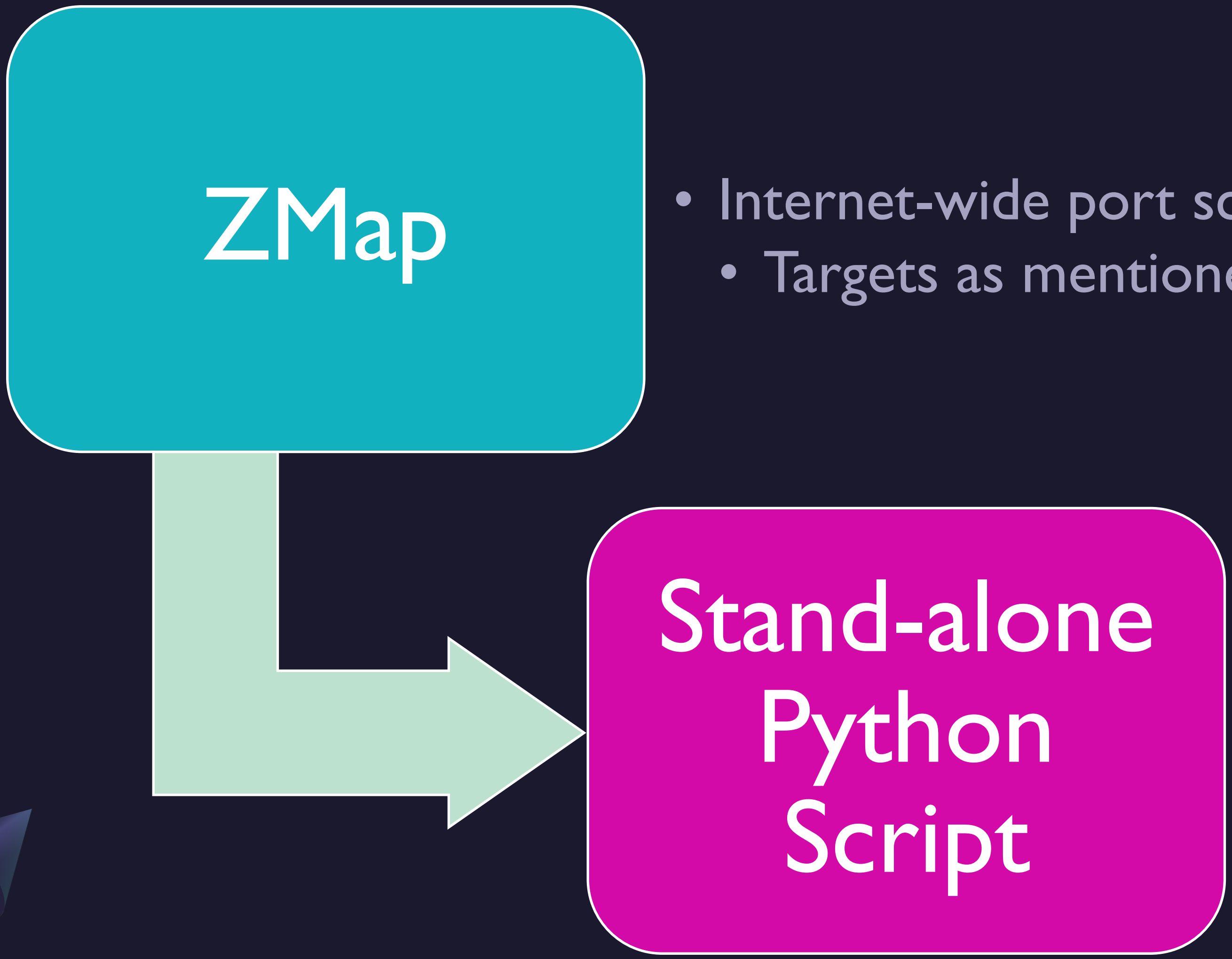
- After the initial handshake
  - The payload will contain the same data structure as the TCP customized packet
  - The *type* field in the header (*struc_proto_header*) will be incremented

Scanner Implementation

- The target protocols/ports were decided based on the recent sample's config values
  - I had to implement the scanner per variant

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)

Scanner Implementation (Cont.)



- Internet-wide port scan
  - Targets as mentioned previously

- Decode the response packet
- Validate the decoded values
  - TCP: payload size fields
  - HTTP(S)/UDP: type and session_dst_id

Multiple Protocol Listening at a Single Port

- One ShadowPad sample config hinted the C2 can accept multiple protocol requests at a single port
- I tested the hypothesis by scanning one active C2

```
[*] 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'
..
```

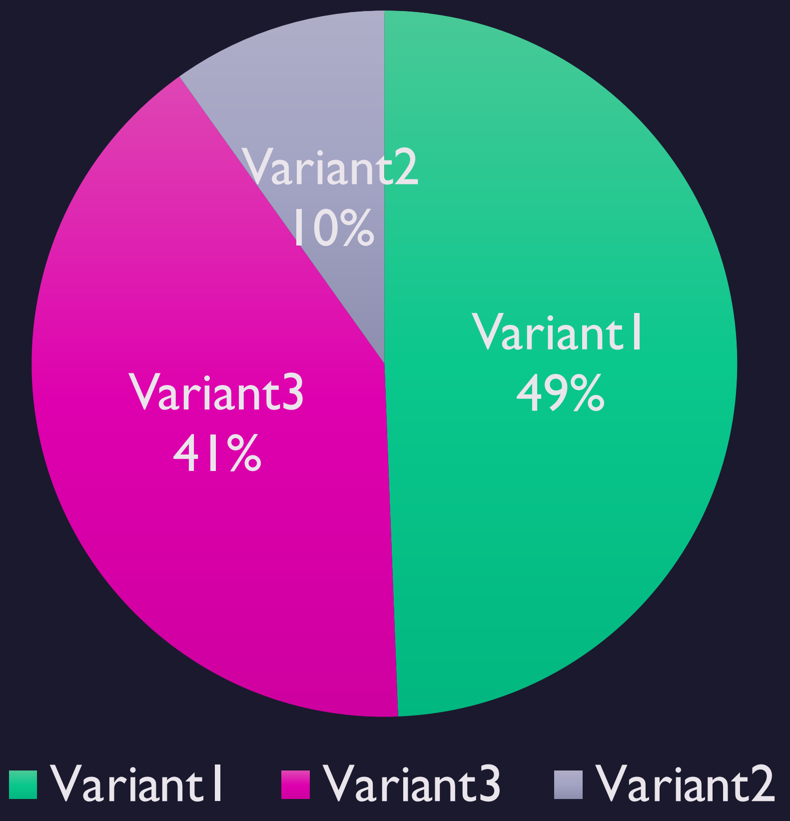
Hostname/port matched

SHA256: d011130defd8b988ab78043b30a9f7e0cada5751064b3975a19f4de92d2c0025

Multiple Protocol Listening at a Single Port (Cont.)

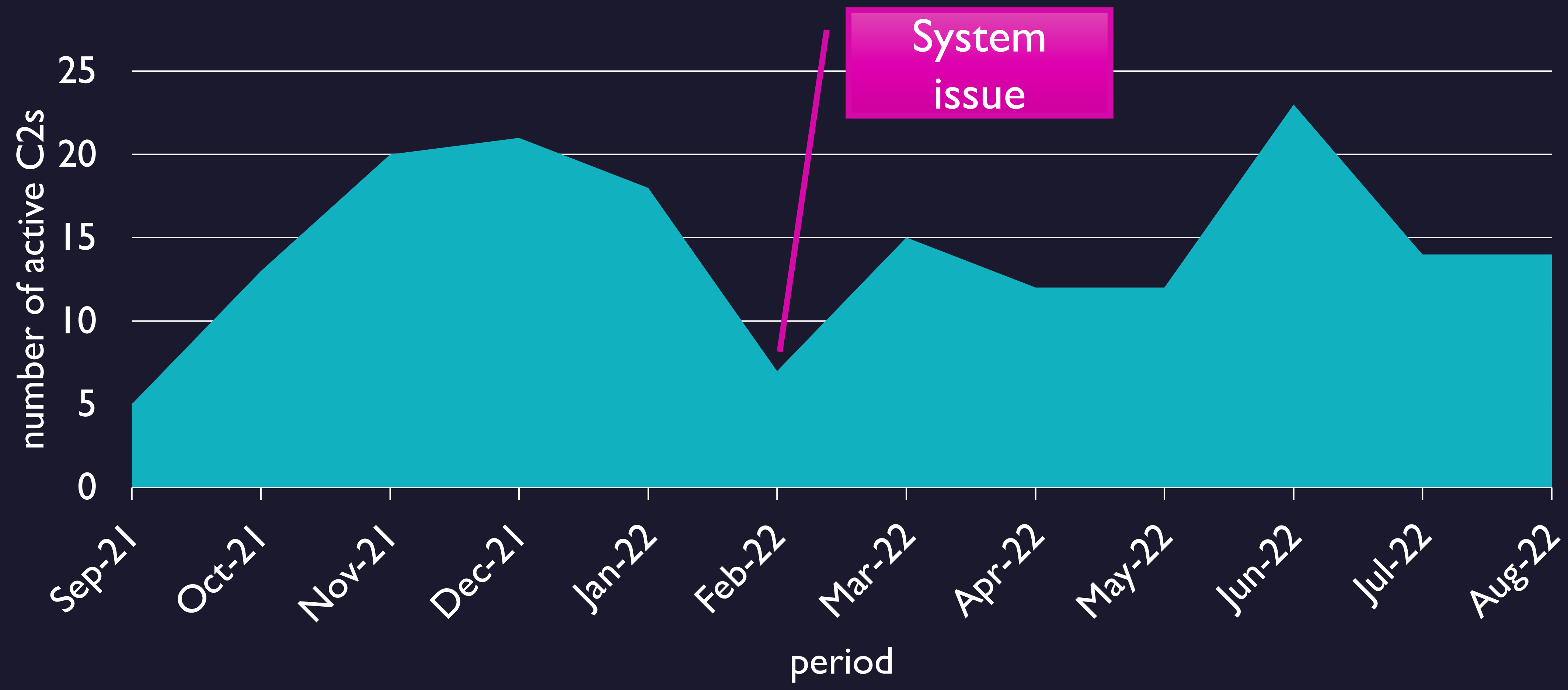
```
$ ./c2fs.py -d -l corpus/query.txt -p 443 -f sp http Variant2
..
[*] malware options: family = ShadowPad; targeted protocol = http (version = Variant2)
[*] ShadowPad specific options: version = Variant2; key size = 2; key endian = big; header size = 0x8; header type = 0x0; client session ID = 53978
[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)
```

Result: Population by Variant



- 2021/09 - 2022/08
- 81 C2 servers, 74 unique IPs
- Variant1 had become more active through the period

Result: Change in Number of Active C2s



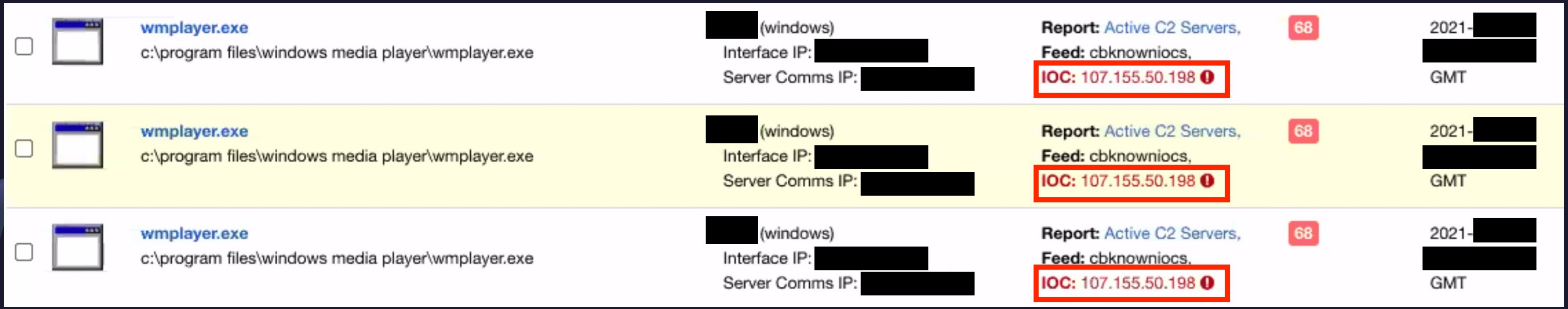
Samples Communicating with C2 IPs

- All C2s were discovered by the TCP/443 Variant1 scanner
  - The C2s tended to accept multiple protocols/ports at the same time
- The scanning system caught the C2s prior to the sample submissions

Sample Malware family	C2 IP address	C2 Protocol/Port used by sample	Sample submission date on VT	C2 first-seen date by scanner	C2 last-seen date by scanner
Spyder	156.240.104.149	TLS/443	2021/10/26	2021/10/16	2021/10/16
ReverseWindow	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

Incident Response Case Triggered by Discovered C2

- The discovery of a ShadowPad C2 IP (107.155.50.198) triggered an incident response
  - The APT attack had bypassed many methods of detection
  - But it was ultimately alerted because of the pre-identified C2 IP



Notes for Internet-wide C2 Scanning

How to Get Input (Port Scan) Data

- I generate input data on my own using ZMap, not purchasing 3rd party data service (e.g., Shodan/CenSys)
  - For UDP-based protocols, we must scan hosts with the customized protocol formats
    - commercial services normally don't provide the option
  - Commercial services don't scan minor ports actively
    - E.g., shodan has published the scanning target ports on the web site

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

Note: The data was collected in 2021/11

Anonymization

- Scanning operations are sometimes forced to be terminated by ISPs and VPS providers
  - In order to sustain the C2 scanning research, the source address should be anonymized
- I utilize one of commercial VPN services

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

Anonymization (Cont.)

- ZMap issue with non-Ethernet interfaces like VPN
- The bug has not been patched yet
  - I recommend to patch using the code explained on the page

segmentation fault when sending IP layer packets #580

Open TakahiroHaruyama opened this issue on Nov 5, 2019 · 1 comment

Wrap-up

Wrap-up

- I've discovered over 130 Winnti and ShadowPad C2s
  - 65% of the IOCs have 0 detections on VT
  - 10 C2s are always active in both
- Little possibility of false positives
  - The C2 protocol formats and encoding are fairly-unique
- The C2 scanning can become a game changer as one of the most proactive threat detection approaches

Acknowledgement

- Tadashi Kobayashi
- Leon Chang
- Brian Baskin

Indicators of Compromise

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)

Questions?

- https://github.com/carbonblack/active_c2_ioc_public