Melting Ice – Tracking IcedID Servers with a few simple steps

By Alex Ilgayev

Published: 2021-05-26 · Archived: 2026-04-05 20:32:59 UTC

Research by: Alex Ilgayev

Introduction

Tracking botnets usually demands a significant amount of effort, time, and threat intelligence know-how. The barrier to entry grows even larger in cases of multi-staged complex malware families such as IcedID, Emotet, and QakBot. Therefore, as malware analysts, we tend to look for ways to automate the process as much as we can — collecting a large scale of samples, identifying them, extracting their configurations, and then having these yield value we are interested in, such as a clearer threat intel picture or a more up-to-date domain reputation engine.

While the malware analysis life is often as difficult as described above, it doesn't *have* to be. Sometimes, if we are clever, 10% of the work will get us 90% of the result we are interested in. In this article, we demonstrate such a sleight-of-hand and show how to hunt IcedID C&C servers quickly and without tracking or analyzing any samples.

IcedID Background

The IcedID banker malware first emerged in September 2017 and has made significant progress since then. This threat, also known as *BokBot*, has constantly been growing in the past year and boasts a wide range of malicious capabilities such as browser hooking, credential theft, MiTM proxy setup and a VNC module, among others.

The bot's internal capabilities, including infection chain and malicious components, were described thoroughly by *Malwarebytes*, *Group-IB*, and *Binary Defense* past publications.

According to the latest infection chain presented by Binary Defense researchers, recent iterations of IcedID contain three malicious components – The entry-point DOC/XLS containing malicious macros; a first stage payload; and finally a second-stage payload, which itself consists of two sub-parts – A 64bit DLL loader, and the encrypted actual bot disguised as a ".dat" file.

Each second-stage payload usually contains 2-4 unique embedded domains, which all resolve to the same IP address. Suppose that we set an objective to track these domains/IPs and block them; naturally, we are interested in the quickest and painless way to do it.

The communication protocol for the victims is HTTPS, so we inspected the bundled TLS certificate. As we can observe below, the IcedID people care more about the hard, concrete guarantees provided by RSA-2048 and less about following security guidelines; the certificate is issued to the defaultly-named "Internet Widgits Pty, Ltd" which resides in "Some State".

Figure 1 – Sample IcedID C&C certificate

If we compare this IcedID certificate with a default self-signed certificate, the most notable difference would be the common-name field. Public servers should have their FQDN (e.g. `checkpoint.com`) in this field and not `localhost` like in this case.

You probably understand where this is going; we promptly got to scanning the wide web and looking for other servers which present this certificate.

Enumerating Servers

Loath to reinvent the wheel, we opted to use the popular internet scanning platform *Censys* to create a list of potential C&C candidates. Additional internet scanning platforms, like *Shodan*, could be fit for this purpose as well.

The Censys engine gave us a great amount of control for certificate querying. After some tweaking and fiddling, we constructed the following query:

```
443.https.tls.certificate.parsed.issuer_dn: "CN=localhost, C=AU, ST=Some-State, O=Internet Widgits Pty Ltd" and 443.https.tls.certificate.parsed.subject_dn: "CN=localhost, C=AU, ST=Some-State, O=Internet Widgits Pty Ltd"
```

Figure 2 – One of Censys results with the certificate

If you doubt the uniqueness of having `localhost` as a common name in a certificate, a short experiment proves it handily: the dozens of servers our search yielded may seem like much, but if you omit the requirement `CN=localhost` from the search query, the results explode in size and number in the ten of thousands.

At the moment, we cannot assume that each of these servers is an active IcedID malicious C&C, so we need to validate them. Luckily, this is possible due to another unique property we discovered during the research.

Validating The Servers

While we were reverse-engineering the bot functionality, we noticed the bot makes a pretty interesting test before communicating with the C&C server. The following code is part of a callback function of `WinHttpSendRequest` and is called before contacting the C&C.

Figure 3 – Certificate verification code

In simple words, this code runs *Fowler–Noll–Vo* 32 bit hash function over the certificate's public key and compares it with the assigned serial number of the certificate. The communication proceeds only when that comparison matches (Or with an XOR-ed value `0x384A2414`).

The *Certificate Serial Number* field is assigned by the *Certificate Authority* and provides a unique identifier for each generated certificate. The Certificate Issuer must ensure that no two distinct certificates with the same Certificate Issuer DN contain the same serial number, but no one can guarantee that.

In our case, the certificate is self-signed, and the malware operators assign its C&C serial number field uniquely according to the above logic. Using this verification algorithm ourselves, against a suspected server, we can make sure it is part of the IcedID infrastructure. We provided a simple python script to verify that logic against a supplied remote server in *Appendix A*.

Applying that method, we discovered that most of the previous potential results *were* IcedID related and narrowed the list down to 52 servers. Given the unique certificate property paired with the uncommon hash function, we could safely deduce that these servers are components in IcedID C&C infrastructure. With this list in hand, we went to take a closer look at the servers.

Server Analysis

By analyzing their internet-facing banners, we could find several similar properties for most of the deployed servers. The most common properties were open ports, operating system, and web-server:

Open Ports – All servers had port 443 available, 94% also listened on port 80 and 77% on SSH port (22).

Operating System – 77% of the servers were running a Debian OS.

Web Server – 94% of the servers were running *nginx* web-server.

We could also see that most of the servers reside in Romania, United States, and Germany:

Figure 4 – IcedID servers locations

Another angle of analysis is via a passive DNS service such as *RiskIQ*. Using this service allows us to find associated domains for each IP address we have and expand our threat intel picture. These extra domains must logically be embedded in some samples that we do not have direct access to. For example, one of the addresses we found was `152[.]89.247.60`, which unresolved to the following IcedID domains:

`formgotobig[.]top`
`pondurovіga[.]top`
`tranmigrust[.]club`
`aswenedo[.]space`

Summary

It is a sad fact of life that doing the “right thing” can backfire. Such is the case for IcedID campaign maintainers here; they produced a self-signed certificate and had their malicious operation support HTTPS, a laudable effort that the vast majority of malware cannot be bothered with, and even some long-running legitimate websites could not be bothered with for the longest time. By doing this, they made it much more challenging to instigate a hostile takeover of their network — but they also made all their servers basically respond to standard scanning services with a cheerful “Hi, I’m a malicious C&C”.

Once these servers were exposed, we were able to continuously track them and analyze their behavior without ever running a regular expression, never mind launching a debugger or disassembler. For someone running a malicious campaign, this is a nightmare scenario; they aim to place the “prize” of the continuously updating C&C server list just out of reach, at the top of the tier list. Instead, due to this too-clever-by-half TLS business, collecting the servers (and obtaining a bunch of juicy information, which is out of scope for this article) became within the analyst’s capabilities with a week of experience. As a wise person once said, think before using a technique, or your opponent will use it against you.

IOC

IcedID C&C servers:

83[.]97.20.249
83[.]97.20.174
194[.]5.249.52
45[.]153.240.135
194[.]5.250.104
152[.]89.247.60
91[.]193.19.97
194[.]5.249.81
83[.]97.20.73
194[.]5.250.35
83[.]97.20.176
194[.]5.250.46
212[.]114.52.186
91[.]193.19.37
45[.]147.231.113
188[.]119.148.75
185[.]38.185.90
45[.]138.172.179
91[.]193.19.51
83[.]97.20.122
194[.]5.249.103
139[.]60.161.63
194[.]5.249.86
139[.]60.161.48
185[.]33.85.35
194[.]5.249.97
79[.]141.164.241
194[.]5.249.90
193[.]109.69.52
194[.]5.249.54
185[.]70.184.87
79[.]141.166.39
146[.]0.77.92
31[.]184.199.11
45[.]129.99.241
194[.]5.249.46
185[.]123.53.202
146[.]0.77.18
31[.]24.228.170
45[.]147.230.88
83[.]97.20.254

45[.]147.230.82
194[.]5.249.72
46[.]17.98.191
45[.]153.241.115
185[.]70.184.41
139[.]60.161.50
194[.]5.249.143
79[.]141.161.176
5[.]149.252.179
45[.]147.228.198
91[.]193.19.251

Appendix A

Testing a server for IcedID certificate:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
import idna

from socket import socket

from OpenSSL import SSL

from cryptography.hazmat.primitives.serialization import Encoding

from cryptography.hazmat.primitives.serialization import PublicFormat

def fnv1a_32(data: bytes) -> int:

"""Fowler–Noll–Vo hash function variation.

Args:

data (bytes): Input data

Returns:

int: Output 32 bit hash

"""

hval_init = 0x811c9dc5
```

```
fnv_prime = 0x01000193

fnv_size = 2 ** 32

hval = hval_init

for byte in data:

hval = hval ^ byte

hval = (hval * fnv_prime) % fnv_size

return hval

def get_certificate(hostname: str, port: int):

"""Connects to the remote server,

and retrieves the certificate.

Args:

hostname (str): Remote hostname

port (int): Remote port (usually 443)

Returns:

Certificate object

"""

hostname_idna = idna.encode(hostname)

# We are building a SSL context on top of a raw socket.

sock = socket()

sock.connect((hostname, port))

ctx = SSL.Context(SSL.SSLv23_METHOD)

ctx.check_hostname = False

# the cert is self-signed so we don't want to verify it

ctx.verify_mode = SSL.VERIFY_NONE

# making SSL handshake

sock_ssl = SSL.Connection(ctx, sock)
```

```
sock_ssl.set_connect_state()

sock_ssl.set_tlsext_host_name(hostname_idna)

sock_ssl.do_handshake()

# retrieving certificate and converting it to an cryptography object

cert = sock_ssl.get_peer_certificate()

crypto_cert = cert.to_cryptography()

sock_ssl.close()

sock.close()

return crypto_cert

def test_is_icedid_c2(hostname: str, port: int) -> bool:

"""Testing whether a remote server is part of IcedID

C&C infrastructure.

Args:

hostname (str): Remote hostname

port (int): Remote port (usually 443)

Returns:

bool: True if the server is IcedID verified, or False otherwise.

"""

try:

# We query the server and retrieve its certificate.

cert = get_certificate(hostname, port)

serial_number = cert.serial_number

# Getting the public key, and hashing it.

public_key = cert.public_key().public_bytes(Encoding.DER, PublicFormat.PKCS1)

fnv_hash = fnv1a_32(public_key) & 0x7fffffff

# Finally comparing the hash to the serial number.
```

if serial_number == fnv_hash or serial_number == fnv_hash ^ 0x384A2414:

return True

except Exception as e:

return False

return False

import idna from socket import socket from OpenSSL import SSL from cryptography.hazmat.primitives.serialization import Encoding from cryptography.hazmat.primitives.serialization import PublicFormat def fnv1a_32(data: bytes) -> int: """Fowler–Noll–Vo hash function variation. Args: data (bytes): Input data Returns: int: Output 32 bit hash """ hval_init = 0x811c9dc5 fnv_prime = 0x01000193 fnv_size = 2 ** 32 hval = hval_init for byte in data: hval = hval ^ byte hval = (hval * fnv_prime) % fnv_size return hval def get_certificate(hostname: str, port: int): """Connects to the remote server, and retrieves the certificate. Args: hostname (str): Remote hostname port (int): Remote port (usually 443) Returns: Certificate object """ hostname_idna = idna.encode(hostname) # We are building a SSL context on top of a raw socket. sock = socket() sock.connect((hostname, port)) ctx = SSL.Context(SSL.SSLv23_METHOD) ctx.check_hostname = False # the cert is self-signed so we don't want to verify it ctx.verify_mode = SSL.VERIFY_NONE # making SSL handshake sock_ssl = SSL.Connection(ctx, sock) sock_ssl.set_connect_state() sock_ssl.set_tlsext_host_name(hostname_idna) sock_ssl.do_handshake() # retrieving certificate and converting it to an cryptography object cert = sock_ssl.get_peer_certificate() crypto_cert = cert.to_cryptography() sock_ssl.close() sock.close() return crypto_cert def test_is_icedid_c2(hostname: str, port: int) -> bool: """Testing whether a remote server is part of IcedID C&C infrastructure. Args: hostname (str): Remote hostname port (int): Remote port (usually 443) Returns: bool: True if the server is IcedID verified, or False otherwise. """ try: # We query the server and retrieve its certificate. cert = get_certificate(hostname, port) serial_number = cert.serial_number # Getting the public key, and hashing it. public_key = cert.public_key().public_bytes(Encoding.DER, PublicFormat.PKCS1) fnv_hash = fnv1a_32(public_key) & 0x7fffffff # Finally comparing the hash to the serial number. if serial_number == fnv_hash or serial_number == fnv_hash ^ 0x384A2414: return True except Exception as e: return False return False

```
import idna

from socket import socket
from OpenSSL import SSL
from cryptography.hazmat.primitives.serialization import Encoding
from cryptography.hazmat.primitives.serialization import PublicFormat

def fnv1a_32(data: bytes) -> int:
    """Fowler-Noll-Vo hash function variation.

    Args:
        data (bytes): Input data
```

```
    Returns:
        int: Output 32 bit hash
    """
    hval_init = 0x811c9dc5
    fnv_prime = 0x01000193
    fnv_size = 2 ** 32

    hval = hval_init
    for byte in data:
        hval = hval ^ byte
        hval = (hval * fnv_prime) % fnv_size
    return hval

def get_certificate(hostname: str, port: int):
    """Connects to the remote server,
    and retrieves the certificate.

    Args:
        hostname (str): Remote hostname
        port (int): Remote port (usually 443)

    Returns:
        Certificate object
    """
    hostname_idna = idna.encode(hostname)

    # We are building a SSL context on top of a raw socket.
    sock = socket()
    sock.connect((hostname, port))
    ctx = SSL.Context(SSL.SSLv23_METHOD)
    ctx.check_hostname = False
    # the cert is self-signed so we don't want to verify it
    ctx.verify_mode = SSL.VERIFY_NONE

    # making SSL handshake
    sock_ssl = SSL.Connection(ctx, sock)
    sock_ssl.set_connect_state()
    sock_ssl.set_tlsext_host_name(hostname_idna)
    sock_ssl.do_handshake()

    # retrieving certificate and converting it to an cryptography object
    cert = sock_ssl.get_peer_certificate()
    crypto_cert = cert.to_cryptography()
    sock_ssl.close()
    sock.close()

    return crypto_cert
```

```
def test_is_icedid_c2(hostname: str, port: int) -> bool:
    """Testing whether a remote server is part of IcedID
    C&C infrastructure.

    Args:
        hostname (str): Remote hostname
        port (int): Remote port (usually 443)

    Returns:
        bool: True if the server is IcedID verified, or False otherwise.
    """
    try:
        # We query the server and retrieve its certificate.
        cert = get_certificate(hostname, port)
        serial_number = cert.serial_number

        # Getting the public key, and hashing it.
        public_key = cert.public_key().public_bytes(Encoding.DER, PublicFormat.PKCS1)
        fnv_hash = fnv1a_32(public_key) & 0x7fffffff

        # Finally comparing the hash to the serial number.
        if serial_number == fnv_hash or serial_number == fnv_hash ^ 0x384A2414:
            return True
    except Exception as e:
        return False
    return False
```

Source: https://research.checkpoint.com/2021/melting-ice-tracking-icedid-servers-with-a-few-simple-steps/