2025 BERLIN

24 - 26 September, 2025 / Berlin, Germany

STEALTH OVER TLS: THE EMERGENCE OF ECH-BASED C&C IN ECHIDNA MALWARE

Yuta Sawabe & Rintaro Koike

NTT Security Holdings, Japan

yuta.sawabe@security.ntt
rintaro.koike@security.ntt

ABSTRACT

In early 2024, while analysing targeted attacks against Japanese organizations, we identified a noteworthy piece of malware, which we named ECHidna. At first glance, ECHidna resembles a standard RAT, but it distinguishes itself by leveraging Encrypted ClientHello (ECH) for its C&C communications.

ECH is a very recent enhancement to TLS, designed to improve privacy by encrypting the ClientHello message, including the Server Name Indication (SNI), which is typically transmitted in plaintext [1].

By utilizing ECH, ECHidna's C&C traffic becomes significantly harder to detect and block using conventional security products. This poses a new challenge for defenders, as visibility into the destination of TLS connections is greatly reduced.

In this paper we begin by outlining the broader campaign in which ECHidna was deployed. We then introduce a detailed technical analysis of the malware, with a particular focus on how it implements C&C communication using ECH. Finally, we explore possible countermeasures against this emerging class of threats, as the abuse of ECH by malware authors is likely to increase in the near future.

By reading this paper you will gain a deeper understanding of ECH and its potential for malicious use, enabling SOC analysts, incident responders and CSIRT members to better defend against threats leveraging this new technology.

CAMPAIGN OVERVIEW

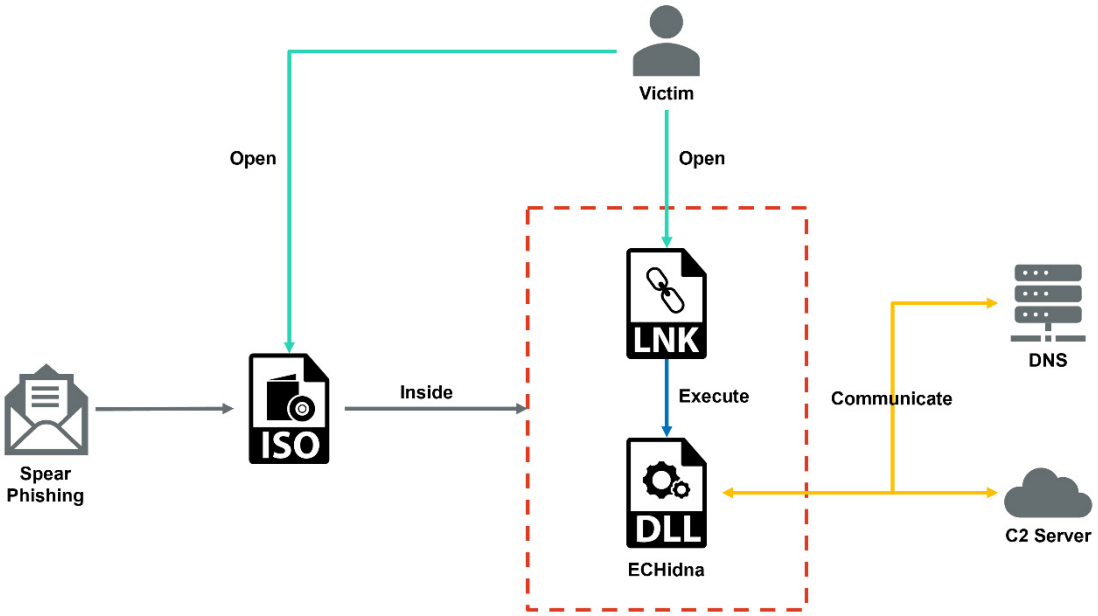

Figure 1: Attack flow.

In May 2024, a spear-phishing email targeting a Japanese research institution was sent. Upon opening the email, the recipient was directed to open an ISO file, which contained several embedded files. Among these was an LNK file that appeared to be a directory. Clicking the LNK file triggered the execution of PowerShell code. Figure 2 shows a reformatted excerpt of that PowerShell code for better readability.

```
$t=$env:TEMP;
$n=$t+'\MS-Service'+$pid+'.dll';
Copy-Item desktop.ini $n;

if(Test-Path $n){
    Start-Process 'rundll32' $n,SSL_version
};
```

Figure 2: PowerShell code executed from the LNK file (excerpt).

This PowerShell script copies a file named `desktop.ini` – a DLL disguised with a .ini extension – from within the ISO to the Temp directory. It then uses `rundll32.exe` to execute the `SSL_version` function of the DLL. This DLL implements an unknown RAT, which we have named 'ECHidna'.

ECHIDNA MALWARE

This section outlines the key behaviours of the DLL file.

When the DLL's `SSL_version` function is invoked, it searches for the marker `BFBFBFBF`, which it uses as a reference point to locate and unpack embedded binary data. The embedded PE file, which is compressed using the Xpress algorithm, is then decompressed in memory via `RtlDecompressBuffer` and executed immediately.

Once the PE file is executed, it establishes a TLS connection with the C2 server using Encrypted ClientHello (ECH) and begins polling the server at regular intervals. If a response is received, the malware parses it to identify a command, executes the corresponding action, and sends the result back to the C2 server.

Command	Function
`setting`	Sends the current malware configuration to the C2 server and saves it locally.
`download`	Downloads a file from a specified URL and reports success or failure to the C2 server.
`upload`	Sends a local file to the C2 server, enabling the exfiltration of host information.
`shell`	Executes a command received from the C2 server through cmd.exe and returns the output.

Table 1: Commands received from the C2 server.

The structure of the data sent to the C2 server for each command is summarized in Table 2.

Field	Description
`cid, id`	Host ID
`cmd`	Command name (setting, download, upload, shell)
`dst`	Identifier of the uploaded file on the C2 server (upload)
`data`	Download result (download)
`result`	Command output (shell) Error code (download) Current malware configuration (setting)

Table 2: Data sent to the C2 server.

The sample generates an eight-digit hexadecimal Host ID to identify the execution environment, based on the following three elements:

- The MAC address of the execution environment
- The file path of the running process (e.g. `C:\Windows\SysWow64\rundll32.exe`)
- A hard-coded string embedded in the sample (e.g. `24508ECH(x86)`, `23830ECH(x86)`)

This Host ID is then included in query parameters and POST data during communication with the C2 server, enabling the server to distinguish between hosts.

By default, the sample contains an initial malware configuration, as shown in Table 3. It can send the current configuration to the C2 server or update it with a new one during communication. When the `setting` command is issued, the malware saves the configuration to the `%ALLUSERSPROFILE%` directory. The filename is based on an eight-digit hexadecimal hash of the executable's file path. Upon subsequent execution, the malware reads this stored configuration file and uses it to establish communication with the C2 server.

Field	Value
`server`	`cloudflera.com[.]ng`
`port`	`443`
`get`	`/webrcs/index.php`
`post`	`/webrcs/upload.php`
`interval`	`3`

Table 3: malware config (default).

The sample also implements several techniques designed to achieve stealth and evade detection.

- Custom Base64 encoding for data obfuscation:

  A custom Base64 table is used to encode response data, POST requests, configuration files and Host IDs. This obfuscation technique helps evade both signature-based detection and network monitoring.

- Dynamic construction of JSON-like custom structures:

  The malware dynamically constructs JSON-like structures in heap memory using a proprietary format, without relying on external libraries. This design makes the data structures harder to parse and complicates static analysis and memory forensics.

- Direct use of low-level *Windows* socket APIs:

  Instead of standard HTTP libraries, the malware uses low-level *Windows* socket APIs – such as `WSAStartup` and `WSAConnectByNameW` – to communicate with the C2 server. This low-level approach helps evade behaviour-based detection by security products.

ENCRYPTED CLIENTHELLO

In traditional TLS, the ClientHello message is sent from the client to the server in plaintext. Consequently, anyone capturing the traffic can observe information such as the destination server (SNI). Encrypted ClientHello (ECH), however, encrypts the ClientHello message, hiding sensitive fields such as the SNI and improving overall privacy.

As of 2025, ECH is supported by *Cloudflare*, *Google Chrome*, *Firefox* and others, meaning that many users may be using ECH-enabled connections without realizing it.

The following section outlines the TLS handshake process involving ECH.

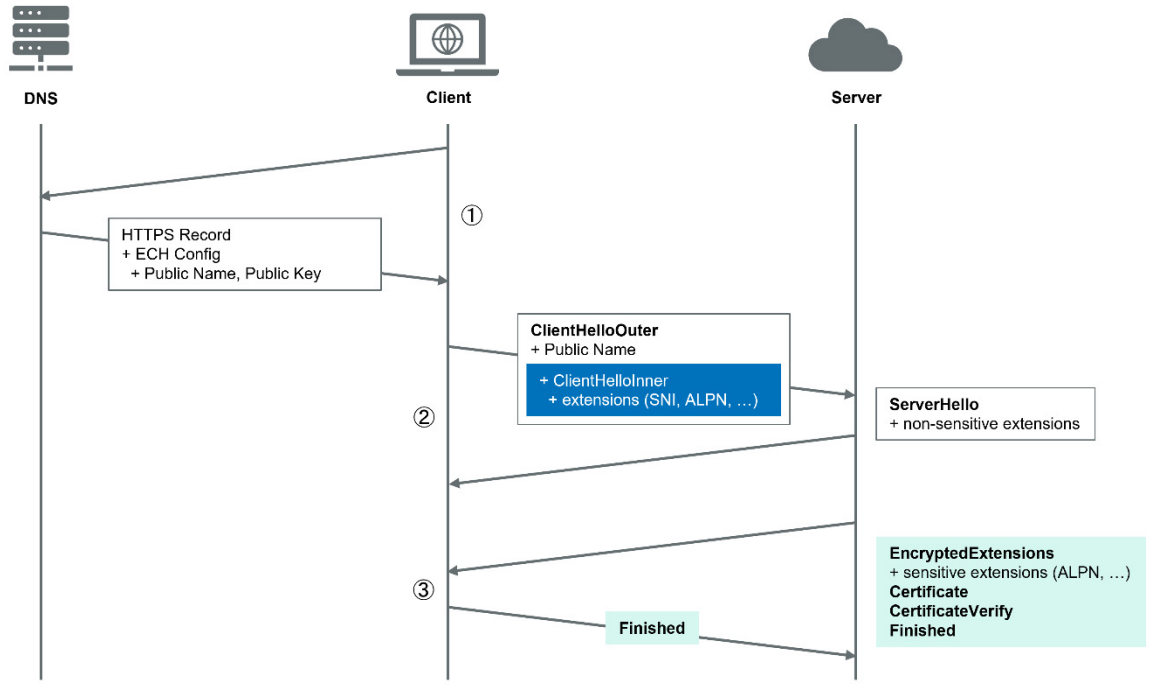

Figure 3: Overview of the TLS handshake process with ECH.

1. Retrieving the ECH config

The client first retrieves the HTTPS record of the target server, typically via traditional DNS, though more privacy-preserving methods such as DNS over HTTPS (DoH) may also be used. For example, the HTTPS record for nao-sec.org contains an 'ech' field embedded within the data section of the response.

```
  "Answer": [
    {
      "name": "nao-sec.org.",
      "type": 65 /* HTTPS */,
      "TTL": 300,
      "data": "1 . alpn=h3,h2 ipv4hint=104.21.27.132,172.67.142.158
ech=AEX+DQBBoQAgACDzjUaCLTKx1ZsfSD/nXtMWBcCxEpVct81frjNDzoVxOwAEAAEAAQASY2xvdWRmbGFyZS1lY2guY29tAAA=
ipv6hint=2606:4700:3031::ac43:8e9e,2606:4700:3033::6815:1b84"
    }
  ],
```

Figure 4: Example of an HTTPS record.

When this field is Base64-decoded and parsed, it yields a structure like the one shown in Table 4.

Field	Value
version	0xfe0d
config_id	0x4f
kem	DHKEM (X25519, HKDF-SHA256)
public_key	ebd0439d07e75dd2e303f997666d1ab0461dd2be896e6b11bf86d7b9babddc2e
ciper_suites	HKDF-SHA256, AES-128-GCM
maximum_name_length	0
public_name	cloudflare-ech.com
extensions	[]

Table 4: Structure of ECH config.

2. Sending the ClientHello

The client uses the retrieved ECH config to construct the outer ClientHello message (`ClientHelloOuter`). `ClientHelloOuter` includes a dummy SNI and an `encrypted_client_hello`, which represents the encrypted form of the inner ClientHello message (`ClientHelloInner`). `ClientHelloInner` includes the actual SNI, ALPN, and other sensitive fields. It is encrypted with the public key from the ECH config and embedded within `ClientHelloOuter`. The final ClientHello message appears as shown in Figure 5.

```
Extension: encrypted_client_hello (len=186)
    Type: encrypted_client_hello (65037)
    Length: 186
    Client Hello type: Outer Client Hello (0)
  Cipher Suite: HKDF-SHA256/AES-128-GCM
      KDF Id: HKDF-SHA256 (1)
      AEAD Id: AES-128-GCM (1)
    Config Id: 226
    Enc length: 32
    Enc: 7316be17ce30ca4cd0df0f7413d588e1356a6f0ff7e1750daf56c279b5d2e706
    Payload length: 144
    Payload […]: aac35be0d964dab6e7401805f6121972836d2bb2dd32ba98a7276839704e
```

Figure 5: Encrypted ClientHello.

3. Server response

Upon receiving `ClientHelloOuter`, the server attempts to decrypt the `encrypted_client_hello`. If decryption succeeds, the server continues the handshake using `ClientHelloInner`. All subsequent communication then proceeds as a standard TLS session. As a result, destination information that would have been visible in plaintext under traditional TLS is now encrypted and concealed from observers.

ECH-BASED C&C

This sample uses ECH to establish a TLS connection to its C2 server, with *Cloudflare* functioning as a reverse proxy. The following describes how this communication is implemented.

1. Retrieving the ECH config

The sample uses DoH to retrieve the HTTPS record for crypto.cloudflare.com and extract the ECH config it contains.

Under normal circumstances, the ECH config should be retrieved from the HTTPS record of the C2 server's domain. However, *Cloudflare*'s ECH service uses a shared ECH config across multiple domains, which is why the sample relies on the HTTPS record of *Cloudflare*'s official domain instead.

The sample resolves domain names by sequentially querying six DoH servers. If a query fails, the sample waits before trying the next server. The wait time increases exponentially with each failure, employing a backoff strategy to avoid detection and maintain stable C2 connectivity. Additionally, the sample tracks the time to live (TTL) of the retrieved ECH config and re-fetches it if expired.

DoH Server URL
https://dns.google.com/resolve
https://doh-2.seby.io/dns-query
https://dns.twnic.tw/dns-query
https://doh-fi.blahdns.com/dns-query
https://doh-jp.blahdns.com/dns-query
https://dns.rubyfish.cn/dns-query

Table 5: DoH server.

2. Sending the ClientHello

The sample uses BoringSSL, a TLS library developed by *Google*, to handle the handshake and ECH-related encryption logic. The generated `ClientHelloInner` includes the C2 server's domain name as the inner SNI. It is then encrypted using the ECH key from the ECH config. The `ClientHelloOuter` includes the encrypted `ClientHelloInner`. It also sets the Public Name from the ECH config (in this case, 'cloudflare-ech.com') as the outer SNI.

```
if ( ech_config && resolve_c2_ip() )
{
  log_message(
    1,
    (int)"C:\\boringssl_x86\\build\\WebRCS\\main.c",
    286,
    "size of esni_key :%d",
    len_ech_config);
  ttl = 0;
  http_header = build_http_header_using_ech();
  host_id = generate_host_id();
  sprintf(url_query, "%s?id=%s", uri_get, (const char *)host_id);
  len_ech_config_1 = (unsigned __int8 *)len_ech_config;
  c2_port = (unsigned __int16)default_port;
  c2_ip = resolve_c2_ip();
  response = send_request(
               c2_domain,
               c2_ip,
               c2_port,
               url_query,
               0,
               http_header,
               (unsigned __int8 *)ech_config,
               len_ech_config_1,
               0,
               0,
               &ttl);
```

Figure 6: Sending the ClientHello to the C2 server.

3. Server response

Upon receiving the `ClientHelloOuter`, *Cloudflare*'s server decrypts the `encrypted_client_hello` and identifies the intended C2 server. It then forwards the connection to the C2 server, acting as a reverse proxy. As a result, the TLS connection is relayed through *Cloudflare* to the C2 server.

DETECTION APPROACH

As previously described, ECH-based communication begins with the retrieval of an ECH config. There are two primary ways to obtain the ECH config:

1. Retrieving the HTTPS record using DNS.
2. Retrieving the HTTPS record using DoH or DoT (DNS over TLS).

In the first case, retrieving the HTTPS record via standard DNS results in plaintext communication. For example, accessing an ECH-enabled website using *Google Chrome* generates a request like the one shown in Figure 7. This serves as valuable evidence for designing detection logic.

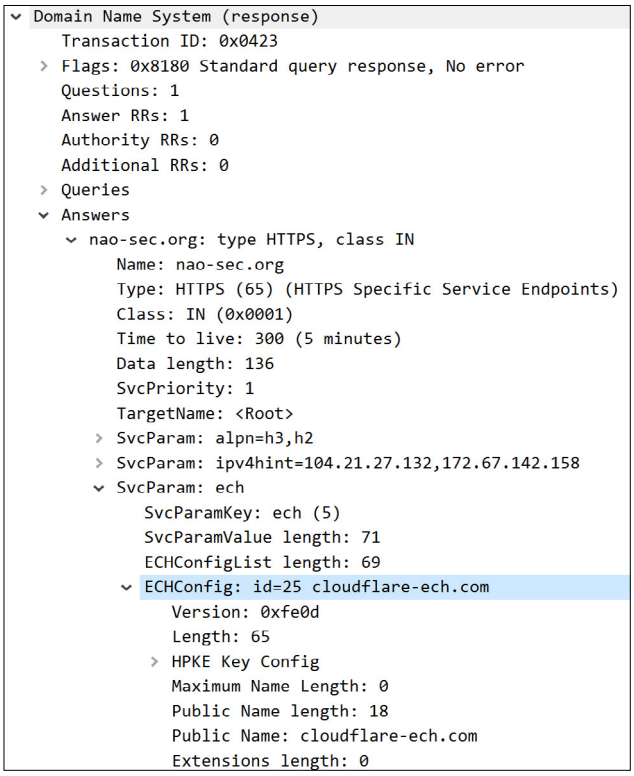

Figure 7: DNS response data.

In contrast, retrieving the HTTPS record via DoH or DoT makes detection significantly more difficult. ECHidna, for example, retrieved the HTTPS record using DoH. Therefore, unless HTTPS traffic is captured and decrypted, detection is extremely difficult.

For domains using *Cloudflare*'s ECH service, the Public Name in the ECH config is typically set to `cloudflare-ech.com`. As a result, all such communication with these domains – whether legitimate or malicious – will appear indistinguishable from traffic to cloudflare-ech.com. This behaviour can serve as an indicator for detecting ECH-based communication. While ECH adoption is gradually increasing, if such communication is observed originating from an environment where normal browsing should not occur, it should be treated as anomalous activity.

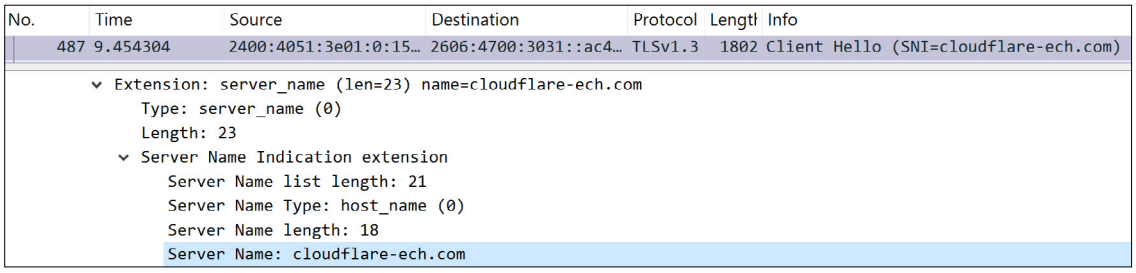

Figure 8: Domains used as Public Name.

That said, detecting malicious behaviour solely from network data remains challenging. A more practical approach is to correlate with endpoint behaviour. In the ECHidna campaign, the most effective detection strategy is to monitor behaviours such as PowerShell execution triggered by an LNK file or the subsequent actions performed by the script.

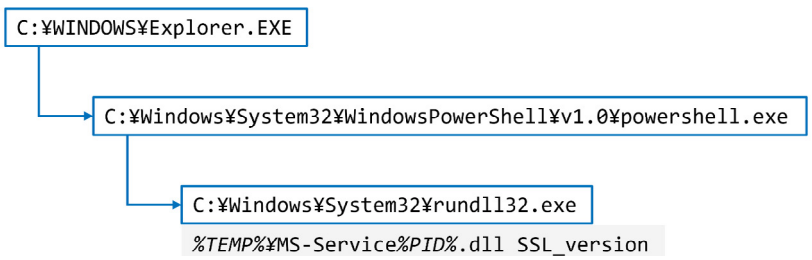

Figure 9: Process tree of ECHidna execution.

CONCLUSION

This paper presented ECHidna, a piece of malware identified in an attack campaign targeting organizations in Japan. ECHidna establishes C2 communication using Encrypted ECH, making the traffic highly difficult to detect. Attackers are actively adopting new technologies. It is therefore essential for members of SOCs, IRs and CSIRTs to stay aware of developments in these emerging technologies and continuously monitor their potential for malicious use.

REFERENCES

[1] IETF. TLS Encrypted Client Hello. https://datatracker.ietf.org/doc/html/draft-ietf-tls-esni.

[2] van der Mandele, A.; Ghedini, A.; Wood, C.; Mehra, R. Encrypted Client Hello – the last puzzle piece to privacy. Cloudflare. 29 September 2023. https://blog.cloudflare.com/announcing-encrypted-client-hello/.

[3] Google Git. BoringSSL. https://boringssl.googlesource.com/boringssl/.

APPENDIX – IOCs

4f7650a2b698db4c95e4ff0f4b6781c9c8f6d00c810892aebbd5b5c54a34b2da

ee8d649c362e75c7d545868c4e1473ebd8d087abf6f916354991d2048eb48025

74aa2eedaa6594efa2075ea2f4617ed3206d228b8fae5fc54382630764bdb5ad

14143e8d8b9e2537db4ee57d86dfd9150641f3c470c66f6d9811743ca0a50441

223f2d2472cad66a06cc1977d0a9d5b99a99c1ba47b458abaa5b629b5da1cdaa