FlashDetect: ActionScript 3 malware detection

Timon Van Overveldt[1], Christopher Kruegel[23], and Giovanni Vigna[23]

[1] Katholieke Universiteit Leuven, Belgium,
timon.vanoverveldt@student.kuleuven.be
[2] University of California, Santa Barbara, USA,
{chris,vigna}@cs.ucsb.edu
[3] Lastline, Inc.

Abstract. Adobe Flash is present on nearly every PC, and it is increasingly being targeted by malware authors. Despite this, research into methods for detecting malicious Flash files has been limited. Similarly, there is very little documentation available about the techniques commonly used by Flash malware. Instead, most research has focused on JavaScript malware.
This paper discusses common techniques such as heap spraying, JIT spraying, and type confusion exploitation in the context of Flash malware. Where applicable, these techniques are compared to those used in malicious JavaScript. Subsequently, FlashDetect is presented, an offline Flash file analyzer that uses both dynamic and static analysis, and that can detect malicious Flash files using ActionScript 3. FlashDetect classifies submitted files using a naive Bayesian classifier based on a set of predefined features. Our experiments show that FlashDetect has high classification accuracy, and that its efficacy is comparable with that of commercial anti-virus products.

Keywords: Flash exploit analysis, malicious ActionScript 3 detection, Flash type confusion

1 Introduction

Adobe Flash is a technology that provides advanced video playback and animation capabilities to developers through an advanced scripting language. The files played by Flash, called SWFs, are often embedded into webpages to be played by a browser plugin, or are embedded into a PDF file to be played by a copy of the Flash Player included in Adobe's Acrobat Reader. The technology is nearly ubiquitous on the desktop: over 99% of all PC users have the Flash browser plugin installed, according to Adobe [1]. However, over the last couple of years, the Flash Player has increasingly become the target of exploitation [18,23,10], with at least 134 high-severity vulnerabilities that have been identified in the Flash Player since January 2009 [15].

Since version 9 appeared in 2006, the Flash Player has supported two scripting languages, ActionScript 2 (AS2) and ActionScript 3 (AS3), each with its own virtual machine. Traditionally, exploits have targeted the older AS2 virtual

machine. However, a number of critical vulnerabilities discovered in the AS3 virtual machine have resulted in an ever growing number of exploits targeting this virtual machine. Even exploits targeting the AS2 virtual machine increasingly turn to AS3 to perform part of the attack. For example, a heap spray might be performed in AS3 before running an AS2 script that exploits an old vulnerability.

Despite the increasing importance of successful solutions to Flash exploit detection, academic research describing such solutions has been scarce. A lot of research has instead focused on JavaScript malware detection. However, without a sound detector for Flash malware, even the most advanced JavaScript malware detector could be circumvented by performing all or part of the attack in Flash.

In this paper, we present FlashDetect, an offline Flash file analyzer and malware detector. FlashDetect combines static bytecode analysis with dynamic analysis using an instrumented version of the Lightspark flash player [19] to enable a high detection rate while maintaining a low false positive rate. The analysis of a Flash file is based on a set of simple yet effective predefined features. These features are used to classify a Flash file using a combination of a naive Bayesian classifier and a single vulnerability-specific filter. FlashDetect is an evolution of OdoSwiff, presented by Ford *et al.* in [6]. However, in contrast to OdoSwiff, FlashDetect focuses solely on the analysis Flash files using AS3, while OdoSwiff mainly covered AS2 exploits. Given the significant differences between AS2 and AS3, we had to develop an entire new set of features and detection techniques. Additionally, OdoSwiff did not employ a naive Bayesian classifier, instead relying solely on threshold-based filters.

The contributions made by this paper include:

- **Insight into common Flash exploit techniques.**
  Techniques commonly used in malicious Flash files, such as obfuscation, heap spraying, and type confusion exploitation are discussed.
- **Detection based on a combination of static *and* dynamic analysis.**
  A hybrid approach to analyzing Flash files is presented, in which the strengths of static and dynamic analysis are combined.
- **Classification based on predefined features.**
  Classification is performed by a combination of a naive Bayesian classifier and a single vulnerability-specific filter. The naive Bayesian classifier is based on a set of predefined features.
- **Evaluation.**
  The merits of our approach are confirmed. Tests performed on 691 benign files and 1,184 malicious files show low false negative and false positive rates of around 1.87% and 2.01%, respectively. These rates are shown to be comparable with or better than those of commercial anti-virus products

The rest of this paper is organized as follows. A number of common techniques employed by ActionScript 3 malware are outlined in Section 2. FlashDetect's implementation details are discussed in Section 3, while Section 4 lists the set of features used for classification. Section 5 presents the experimental results. Finally, FlashDetect's limitations are discussed in Section 6, while Section 7 discusses related publications.

2 Common Flash exploit techniques

This section provides a brief overview of some common techniques employed by Flash malware. Where applicable, comparisons are made with techniques common to JavaScript malware.

2.1 Obfuscation

Obfuscation is often employed by malicious JavaScript or shellcode, and malicious Flash files are no exception. Obfuscation techniques differ according to the technology available for obfuscation. For example, JavaScript is an interpreted language lacking a bytecode representation. Consequently, obfuscation in JavaScript often consists of identifier mangling and/or repeated calls to `eval()`. Shellcode obfuscation, on the other hand, is often achieved by packing the binary data in a specific format, possibly using the XOR operation

We have examined hundreds of malicious Flash files to determine if and how they perform obfuscation, and to develop ways of detecting such obfuscation. The description that follows is the result of that effort.

Flash files are created by compiling ActionScript 3 scripts into a bytecode representation and wrapping that bytecode in an SWF container. Because the ActionScript 3 virtual machine interprets bytecode, a JavaScript-style `eval()` function is not supported. Consequently, Flash obfuscation techniques have more in common with obfuscation techniques for shellcode than with those for JavaScript.

Note that we distinguish two types of obfuscation. The first type is source-code level obfuscation (e.g. identifier renaming). This type of obfuscation is heavily used in JavaScript malware, but given that ActionScript is distributed in bytecode form, it is less prevalent in Flash malware. The second type of obfuscation consists of multiple levels of embedded code. Since our detector will analyze the bytecode of a Flash file, we are most interested in the latter form of deobfuscation.

Though ActionScript 3 does not support `eval()`, it does support the runtime loading of SWF files. This is achieved by calling `Loader.loadBytes()` on a `ByteArray` containing the SWF's data. Using the `DefineBinaryData` SWF tag, arbitrary binary data can be embedded into an SWF file. At runtime, the data becomes available to ActionScript in the form of a `ByteArray` instance.

The `DefineBinaryData` SWF tag is often used in combination with the `Loader.loadBytes()` method to implement a primitive form of obfuscation. However, given that static extraction of `DefineBinaryData` tags is fairly easy using a range of commercial or open-source tools, obfuscation almost never stops there. Instead, malicious Flash files often encode or encrypt the embedded binary data and decode it at runtime before calling `Loader.loadBytes()`.

As is the case with JavaScript [8], obfuscation is actively used by both benign and malicious Flash files, as evidenced by commercial obfuscators such as DoSWF [5] and Kindi [11]. Thus, the mere presence of obfuscated code is not a good indicator of the maliciousness of a Flash file. Therefore, a need for a

dynamic extraction method arises, allowing static analysis to be performed after deobfuscation. To this end, we have modified the Lightspark flash player so that each time `Loader.loadBytes()` is called, the content of the `ByteArray` is dumped into a file for later analysis. This allows for reliable extraction of embedded SWFs, as long as the deobfuscation code path is reached. In cases where the deobfuscation code path is not reached and no files are dynamically extracted, we fall back to a simple static extractor.

2.2 Heap spraying

Heap spraying is an extremely common technique found in contemporary malware, and as such, it is commonly employed in ActionScript 3 malware. As is the case with obfuscation techniques, the way a heap spray is performed depends on the environment in which it needs to be performed. For example, in JavaScript, heap sprays are often performed by creating a string, and repeatedly concatenating the string with itself. In ActionScript 3, the most common way to perform a heap spray is through the use of a `ByteArray`.

The `ByteArray` class available in ActionScript 3 provides byte-level access to a chunk of data. It allows reading and writing of arbitrary bytes, and also allows the reading and writing of the binary representation of integers, floating point numbers, and strings. The implementation of the `ByteArray` class in the ActionScript 3 virtual machine uses a contiguous chunk of memory that is expanded when necessary to store the contents of the array. Therefore, the `ByteArray` class is a prime candidate for performing heap sprays.

Heap spraying code often uses one `ByteArray` containing the shellcode to be written, and a second `ByteArray` to perform the actual heap spray on. A simple loop is then used to repeatedly copy the first array's contents into the second. This results in the second array's memory chunk becoming very large, covering a large portion of the process's memory space with shellcode. Another common way to perform a heap spray is to use a string that contains the hexadecimal, base64, or some other encoding of the shellcode. This string is then decoded before being repeatedly written into a `ByteArray`, until it covers a large portion of the memory space.

2.3 JIT spraying

The concept of JIT spraying in ActionScript 3 has been introduced in [2]. In that paper, the author shows how the JIT compiler in the ActionScript 3 virtual machine can be forced to generate executable blocks of code with almost identical semantics to some specified shellcode. It is shown that a chain of XOR operations performed on a set of specially crafted integers is compiled to native code in such a way that, when the code is jumped into at an offset, it is semantically equivalent to the given shellcode. The concept of JIT spraying is significant because it allows bypassing the *Data Execution Protection (DEP)* feature present in most modern operating systems.

We have observed that exploits targeting the Flash Player and using JIT sprays are fairly common. It seems that the most common way in which a JIT spray is performed consists of repeatedly loading an embedded SWF file containing the bytecode that performs the repeated XOR operations. This repetition ensures that multiple blocks of executable shellcode are present in the memory of the Flash player. Furthermore, the shellcode itself often contains a NOP-sled to further enhance the chances of successful exploitation.

2.4 Malicious ActionScript 3 as exploit facilitator

Although a range of vulnerabilities have allowed the ActionScript 3 virtual machine to be a target for exploitation, another common use of Flash malware seems to be that of a *facilitator* of other exploits. We have observed instances where a malicious Flash file merely performs a heap spray, without trying to gain control of execution. This seems to indicate that the malicious file is meant to facilitate another exploit, for example one targeting a JavaScript engine. The rationale behind this behavior is that Flash files are often embedded in other resources, and as such, they can facilitate exploits targeting the technologies related to those resources.

For example, exploits targeting a JavaScript engine might use ActionScript 3 to perform a heap spray, after which the actual control of execution is gained through a vulnerability in the JavaScript engine. However, the value of such types of exploits is declining, as browsers are increasingly separating the browser from its plugins by running those plugins in separate processes. Another possible scenario in which a malicious Flash file acts as an exploit facilitator is that in which a Flash file is embedded in a PDF. In such a case, the Flash file might perform a heap spray, while the actual control of execution is gained through a vulnerability in the JavaScript engine in Adobe's Acrobat Reader. Finally, as mentioned in the previous section, ActionScript 3 is often used to facilitate exploits targeting the ActionScript 2 virtual machine.

These examples illustrate that Flash is a versatile tool for malware authors, as Flash files can be used both to launch full-fledged attacks, as well as to act as a facilitator for the exploitation of other technologies.

2.5 Type confusion exploitation

A relatively recent development has been the exploitation of type confusion vulnerabilities present in both the ActionScript 2 and ActionScript 3 virtual machine. These types of exploits are interesting since they often allow an attacker to construct a very reliable exploit that completely bypasses both *Data Execution Protection* and *Address Space Layout Randomization*, without relying on heap or JIT spraying. There are a number of vulnerabilities for which the exploit code, or at least parts of it, have been published. Among these are CVE-2010-3654 and CVE-2011-0609, which relate to the ActionScript 3 virtual machine, and CVE-2011-0611, which relates to the ActionScript 2 virtual machine. Other

Listing 1.1. Implementation of the 3 classes used to exploit CVE-2010-3654.

```
1  class OriginalClass {
2      static public function getPointer (o:Object):uint { return 0; }
3      static public function tagAsNumber (u:uint):* { }
4      static public function fakeObject (p:uint):SomeClass { return null; }
5  }
6
7  class ConfusedClass {
8      static public function getPointer(o:Object):Object { return o; }
9      static public function tagAsNumber(p:uint):uint { return p | 0x7 }
10     static public function fakeObject(p:uint):uint { return p; }
11 }
12
13 class SomeClass {
14     public function someMethod():* {}
15 }
```

security advisories, such as CVE-2012-0752, are also known to relate to type confusion vulnerabilities, although their exploit code is not public yet.

An exploit for CVE-2010-3654 is presented by Li in [13]. We present a slightly simplified adaption of this exploit, to illustrate the way in which a type confusion exploit works.

ActionScript 3 virtual machine implementation The ActionScript 3 virtual machine operates on data types called 'atoms.' Atoms consist of 29 bits of data followed by a 3-bit type-tag. The data can either be an integer or an 8-byte-aligned pointer to some larger data (as is the case for atoms representing objects or floating point numbers). The type-tags allow the virtual machine to support runtime type detection when a variable's type is not specified in the source code.

The ActionScript 3 virtual machine also contains a JIT compiler that compiles ActionScript 3 methods to native code. The native code for such a method works solely with native data types, not atoms. Thus, native code methods return ActionScript objects as pointers, integers as 32-bit integers, and floating point numbers as pointers to IEEE 754 double precision numbers. Code calling a native code method then wraps the result into a type-tag to form an atom, so that the result can be used by the virtual machine. The type-tag that is used when wrapping native code results depends on the method's return type, which is specified by the class definition.

Elements of a CVE-2010-3654 exploit. An exploit for CVE-2010-3654 consists of three classes. Listing 1.1 lists the implementation of those three classes. To trigger the vulnerability, the list of identifiers in the compiled bytecode is modified such that the name of `ConfusedClass`'s identifier is changed

to `OriginalClass`. After this modification, the list of identifiers will have two entries named `OriginalClass`. The result is that in vulnerable version of the Flash player, the `ConfusedClass` is 'type confused' with the `OriginalClass`.

When `ConfusedClass` is confused with `OriginalClass`, calls to methods of `OriginalClass` instead end up calling the native code implementations of `ConfusedClass`'s corresponding methods. However, when wrapping the results of these native code methods, the type-tag that is used depends on the return types defined in `OriginalClass`. This mismatch, where on the one hand `ConfusedClass`'s native code methods are called, while on the other hand `OriginalClass`'s return types are used, results in an exploitable vulnerability.

The following sections show how this vulnerability can be used to leak addresses of objects, read arbitrary memory and gain control of execution.

Leaking objects' memory addresses. Because of the type confusion, the pointers to the `Object`s returned by `ConfusedClass.getPointer` are wrapped in `uint` type-tags (as specified by `OriginalClass`), exposing the pointer to the ActionScript 3 runtime. For example, `OriginalClass.getPointer(new ByteArray())` will actually call `ConfusedClass.getPointer` and return the memory address of the `ByteArray` in the form of an integer.

Reading arbitrary memory addresses. Similarly, the integers returned by `ConfusedClass.tagAsNumber` end being used as if they are atoms with a type-tag. This is because the type of the returned value needs to be inferred at runtime, as no type is specified in `OriginalClass`. The 0x7 type-tag that is added by `ConfusedClass.tagAsNumber` is that of a floating point number atom.

In the ActionScript 3 virtual machine, the data of a floating point number atom is an 8-byte-aligned pointer to an IEEE 754 double precision number. Thus, after a call to `tagAsNumber`, the given integer will be used as if it is an 8-byte-aligned pointer pointing to valid IEEE 754 data.

This effectively allows an attacker to read arbitrary memory locations by passing the memory location to `tagAsNumber` and then writing the 'fake' floating point number to a `ByteArray`. This results in the 8 bytes at the given location being written into the `ByteArray`. These bytes can then be read separately using the methods provided by the `ByteArray` class.

Gaining control of execution. Finally, by passing a memory address to the `fakeObject` method, one can create a 'fake' object of type `SomeClass` whose representation is *supposedly* stored at the given memory location. When `someMethod` is called on this fake object, the virtual machine will access the object's vtable at a certain offset from the given memory address, look up the method's address, and then jump to it. Thus, by specifically crafting the memory at the given memory location, the attacker can make the virtual machine hand over control of execution to a piece of memory under his control. Crafting such a chunk of memory can easily be done by using a `ByteArray` instance and then leaking

the address of that `ByteArray`'s data using a combination of the previous two techniques.

Bypassing DEP. It is clear that by being able to leak pointers to objects, read arbitrary memory addresses and gain control of execution, one can easily create a basic exploit that bypasses ASLR. However, DEP still prevents such exploits from working, since any shellcode produced in ActionScript will be non-executable.

There is, however, an important element that circumvents this last hurdle: the first 4 bytes of an ActionScript object's representation point to a memory address in the Flash player DLL that is always at constant offset to the start of the DLL. Thus, by reading these 4 bytes, an attacker can infer the base address of the Flash player DLL in the process's memory.

Since the DLL contains a call to `VirtualProtect`, an attacker can use the information to discover the address of that function. Therefore, all an attacker needs to do to circumvent DEP is to create a return-oriented programming attack that calls `VirtualProtect` on some shellcode, using the gadgets available in the DLL. Afterwards, the shellcode, *which can be arbitrarily large*, can be jumped to and executed.

2.6 Environment-identifying code

Environment-identifying code reads some property and compares it to some constant. The result is then used in a conditional branch. Such code is often used to selectively launch an exploit only if the Flash file is being executed by a vulnerable version of the Flash player. Therefore, a malicious Flash file might not exhibit distinctive behavior if such environment-identifying code fails to identify a vulnerable Flash player instance. From our observations we have concluded that environment-identifying code is fairly common in malicious Flash files. Since environment-identification is often used to target specific versions of the Flash runtime, it is not possible to instrument the Lightspark player to return a single version string that is vulnerable to all exploits.

To improve the detection rate and to maximize the chances of successfully deobfuscating any embedded SWF files, we have instrumented the Lightspark player to taint all environment-identifying properties. More precisely, all properties of the `Capabilities` class are tainted. These taint values are propagated through all basic ActionScript 3 operations (e.g., string concatenation). Subsequently, whenever an `ifeq` ('if equals') branch depends on a tainted value, that branch is forcibly, taken no matter what the actual result of the comparison operation is. On the other hand, complementary `ifne` ('if not equals') branches depending on tainted values are never taken.

Listing 1.2 contains a simplified excerpt of a real malicious sample performing environment-identification. The sample matches the Flash player's version (obtained from the `Capabilities.version` property) to a predefined set of versions. Each version has an accompanying embedded SWF file containing an exploit targeting that version. With the original Lightspark player, chances are high that

Listing 1.2. Excerpt of a sample using environment-identifying bytecode

```
1      getlex          flash.system::Capabilities
2      getproperty     version
3      coerce          String
4      setlocal        15
5      getlocal        15
6      pushstring      "WIN 9,0,115,0"
7      ifne            L1
8
9      // load exploit targeting Windows Flash Player version 9.0.115
10
11 L1: getlocal        15
12     pushstring      "WIN 9,0,16,0"
13     ifne            L2
14
15     // load exploit targeting Windows Flash Player version 9.0.16
16
17 L2: // this Flash player is not vulnerable, do not load any exploit
```

both `ifne` conditional branches would be taken, resulting in the malicious Flash file not launching any exploit. However, because of the use of taint-propagation, none of the `ifne` branches are taken in our analysis environment, resulting in the last exploit being loaded.

Note that this simple approach works well, as most environment-identifying code uses inclusive rules to determine vulnerable Flash players. That is, instead of determining that a given Flash player instance *is not* vulnerable, most environment-identifying code determines that a given instance *is* vulnerable. However, it is clear that this approach can be circumvented, a scenario that is discussed in more detail in Section 6.

3 Detector implementation

FlashDetect's analysis of a Flash file is split into three phases. In the first phase, the Flash file is dynamically analyzed using an instrumented version of the Lightspark flash player. The second phase leverages a static analysis of the ActionScript 3 bytecode of the Flash file (including any bytecode found through deobfuscation during the dynamic analysis phase). Finally, in the last phase, the Flash file is classified using the set of features described in Section 4.

3.1 Phase I: dynamic analysis

The dynamic analysis of submitted Flash files is performed by an instrumented version of the Lightspark flash player that saves a trace of interesting events, such as the calling of methods, access to properties, or the instantiation of classes.

After the dynamic analysis, this trace is analyzed to determine the values of the features used for classification.

Additionally, the instrumented Lightspark version saves every SWF file that is loaded at runtime. This allows the subsequent static analysis to take into account both the original SWF file and any other embedded, possibly obfuscated, SWF files that were loaded at runtime.

During the dynamic analysis, as soon as the first bytecode instruction is executed, a timer is started with a given timeout value. When the timer runs out, the dynamic analysis ends. This way, each Flash file is analyzed for more or less the same amount of time. At the same time, starting the timer only after the first instruction has executed ensures that the time spent loading a Flash file has no effect on the actual amount of time for which the file's scripted behavior is analyzed.

3.2 Phase II: static analysis

After performing the dynamic analysis, we perform a static analysis of both the original SWF file as well as any deobfuscated SWF files. The majority of the static analysis phase is spent analyzing the ActionScript 3 bytecode found in the SWFs. However, a small part of the static analysis checks for commonly exploited vulnerabilities in the Flash player's SWF parser, such as integer overflows in SWF tags.

As is the case with the dynamic analysis, the goal of the static analysis is to determine the values for the set of features used for classification. Some feature values are determined during both the dynamic and static analysis phase. For example, the feature that captures a Flash file accessing the `Capabilities.version` property, commonly used for environment-identification purposes, is detected during both analysis phases.

Checking the same feature during both phases might seem redundant, but it has a practical advantage. As the Lightspark flash player is a fairly recent project, it does not yet run all Flash files correctly. Hence, it is possible that the dynamic analysis phase will be cut short because of an unexpected error. However, because the same feature is also checked in the static analysis phase, chances are high that the feature will still be correctly detected. On the other hand, since it is possible to obfuscate property or method names, static analysis might fail to detect certain features. However, as long as the dynamic analysis succeeds, that feature will still be detected correctly.

3.3 Phase III: classification

A naive Bayesian classifier is used to classify submitted samples using the set of features mentioned earlier. The classifier accepts features with both a Boolean value domain and a continuous value domain. A Laplacian correction is applied to Boolean features to convert zero-probabilities to very small probabilities.

As is common with naive Bayesian classifiers, the normal distribution is used to model continuous features. Thus, the probability of continuous features is estimated using the normal probability density function.

$$f(x, \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} exp\Big(\frac{-(x-\mu)^2}{2\sigma^2}\Big)$$

Additionally, calculated probabilities are clamped to a minimum probability of 1e-10 , so as to prohibit any single feature from influencing the classification decision too much.

Finally, there is one feature that we consider a definitive indicator of maliciousness, and for which we bypass the naive Bayesian classifier. This feature is discussed in Section 4.

4 Features used for classification

Our features can be grouped according to the type of behavior they characterize. Boolean features are marked by *(B)*, while continuous features are marked by *(C)*.

Features related to embedded and obfuscated SWF files or shellcode. The group consists of the following features:

- **shellcode (B):** indicates whether the `sctest` tool from the *Libemu* library detected shellcode embedded in the Flash file.
- **load-loadbytes, loader-load (B):** indicate whether the Flash file uses the `loadBytes` or `load` methods of the `Loader` class to load SWF files at runtime.
- **obfuscation-method-ratio (C):** represents the ratio of deobfuscation-related method calls to the overall number of method calls; the methods are `fromCharCode()`, `charCodeAt` and `slice()` of the `String` class, in addition to `parseInt`. These methods are frequently used in deobfuscation in the wild.
- **bytearray-method-ratio (C):** the ratio of `ByteArray`-related method calls to the total number method calls. A large ratio is indicative of deobfuscation and/or heap spraying.
- **bytearray-callprop-ratio (C):** the ratio of `ByteArray` method-calling bytecode instructions to the total number method-calling instructions. A large ratio is indicative of deobfuscation and/or heap spraying.
- **avg-pushstring-char-range (C):** the average range of characters in `pushstring` instructions; indicative of strings containing binary data such as shellcode.
- **avg-base64-pushstring-length (C):** the average length of strings pushed by `pushstring` instructions that match the base64 character set; indicative of obfuscated data.

Of the above features, the *shellcode*, *bytearray-callprop-ratio*, *avg-pushstring-char-range* and *avg-base64-pushstring-length* features are only checked during the static analysis phase. The *obfuscation-method-ratio* and *bytearray-method-ratio* features are only checked during the dynamic analysis phase. The *loader-loadbytes* and *loader-load* features are checked during both phases of the analysis.

Features related to environment-awareness.

- **checks-url (B):** indicates whether the Flash file checks the URL of the webpage it is embedded in; the feature indicates access to the `LoaderInfo.url` property or access through the `ExternalInterface` class to the `window.location` property of the embedding HTML page. Most malicious files *do not* check the URL, while a lot of benign files *do*, so this feature is indicative of benign behavior rather than malicious behavior.
- **external-interface (B):** indicates whether the Flash file uses any of the methods of the `ExternalInterface` class to communicate with the external host of the Flash plugin, such as the web browser. Most malicious files *do not* use `ExternalInterface`, while a lot of benign files *do*, so this feature is indicative of benign behavior rather than malicious behavior.
- **checks-capabilities (B):** indicates whether the Flash file uses any of the properties of the `Capabilities` class; indicative of environment-identification often used by malicious files in the wild. Additionally, there are five features that specifically indicate access to the `version`, `playerType`, and `isDebugger` properties of the `Capabilities` class.

All features in this group are checked during both analysis phases.

Features related to general runtime behavior. The features in this group all provide an indication how a given Flash file behaves in very general terms. For example, a very high amount of method calls or high ratio of `push` opcodes is often indicative of deobfuscation or heap spraying.

- **method-calls-per-second, method-calls-per-cpusecond (C):** the number of built-in methods (provided by the Flash Player runtime) called during the dynamic analysis phase, normalized by the running time in terms of wall clock or CPU time, respectively. These features each have another variant in which the value is additionally normalized by the size of the bytecode present in the SWF file.
- **avg-*opcode*-ratio (C):** a group of features that indicate the ratio of a certain opcode to the total number of opcodes. The opcodes for which ratios are determined are `bitxor`, the `push` opcodes used to push data on the stack, and the `call` opcodes used to call methods.

The values to the *avg-opcode-ratio* features are determined during the static analysis phase, while the other features' values are determined during the dynamic analysis phase.

Vulnerability-specific features.

- **bad-scene-count (B):** indicates whether an invalid `SceneCount` value was detected inside the `DefineSceneAndFrameLabelData` SWF tag. This is indicative of an exploit targeting the CVE-2007-0071 vulnerability.

The value to this feature is determined during the static analysis phase.

This feature is used because some malicious files only trigger the CVE-2007-0071 vulnerability, without containing any other malicious content or bytecode. These files are probably meant to be embedded inside other files that do contain malicious content (e.g. shellcode), to act as exploit triggers.

Given that these types of malicious files would not be detected by the other features, we use this vulnerability-specific feature, enabling a fair comparison with other detection products. Given that the presence of this feature is such a definitive sign of maliciousness, we immediately consider files exhibiting the feature to be malicious, bypassing the naive Bayesian classifier, as mentioned earlier in Section 3.3.

5 Evaluation

5.1 Sample selection

To evaluate the accuracy of FlashDetect, we tested the classifier on a set of Flash files that are known to be malicious or benign.

Benign samples. The benign samples were manually verified to be benign and were gathered by crawling the following sources:

- Miniclips.com, an online games website.
- Various websites offering free Flash webdesign templates.
- Google search results for the query `filetype:swf`.

Additionally, the benign sample set includes Flash files submitted to Wepawet that, after manual verification, were found to be benign. In total, the benign sample set consists of 691 Flash files.

Malicious samples. The malicious samples were gathered from files submitted to Wepawet, and they were manually verified to be malicious. Hence, the samples were categorized according to their similarity, as shown in Table 1. The table also shows whether or not the files in each category contain embedded/obfuscated SWF files, and whether or not the environment-identifying `Capabilities` class is accessed. Note that 7 out of 12 categories use embedded, possibly obfuscated, SWFs, while 4 out of 12 categories perform environment-identification.

Table 2 shows the fraction of benign and malicious files that access the different properties of the `Capabilities` class that are used for environment-identification, and that are also used as features for classification.

Table 1. Categorization of malicious samples with the number of samples. Also shown is whether or not embedded SWF files are used, and whether or not the environment-identifying `Capabilities` class is accessed.

Group	Number of samples	Embedded files	`Capabilities` access
zasder	1016	✓	✓
corrupt-obfuscated-avm1	49	✓	✓
uncategorized	24	✓	✓
loaderinfo-parameters-sc	16		
pop	15		
hs	14	✓	
woshi2bbd-jitspray	11	✓	✓
jitegg	10	✓	
flashspray	9		
badscene	9		
doswf-sc1	6	✓	
heapspray-1	5		
Total	1184		

Table 2. Fraction of files that access the different properties of the `Capabilities` class.

	`isDebugger`	`playerType`	`version`
Benign files	0.330	0.378	0.421
Malicious Files	0.003	0.872	0.870

Finally, some of the categories are notable for the way the exploits work.

- **Zasder.** Contains files that employ environment-identification and multiple levels of obfuscation, and eventually try to exploit CVE-2007-0071 [14].
- **Corrupt-obfuscated-avm1**. Contains files that load an obfuscated ActionScript 2 Flash file with a corrupt structure that presumably triggers some vulnerability in the Flash player. As such, these files are good examples of ActionScript 3 being used as an ActionScript 2 exploit facilitator.
- **Woshi2bbd-jitspray.** These files repeatedly load an obfuscated SWF that contains JIT spraying bytecode, after which a final SWF is loaded that presumably tries to exploit some vulnerability.
- **Flashspray.** These files only call a JavaScript function called `FLASHSPRAY` in the HTML page embedding the file. They are presumably used to circumvent JavaScript malware detectors. Again, these files are good examples of malicious ActionScript 3 in an exploit facilitating role, this time probably facilitating an exploit targeting a browser vulnerability.

5.2 Experimental results

To test the efficacy of the classifier, we hand-picked a set of training samples from the malicious samples. This set of 47 samples includes at least one sample from

each category. This way, no single category is over-represented in the training data, even if that category has many more samples than the others. For the benign training data, we randomly selected 47 benign samples. The classifier was then trained on the training samples, and tested on the remaining 1781 samples. We repeated this test 20 times, each time with a different set of randomly selected benign training samples.

In addition to testing the classifier using manually selected malicious training samples, we tested the classifier using randomly selected samples. For this purpose, we partitioned the randomized malicious samples into 20 disjunct sets. Subsequently, we used each set in turn as the malicious training sample set. An equal number of benign training samples accompanying the malicious training samples were again randomly selected. By the pigeonhole principle, this setup ensures that for all but the three largest malicious categories, there is at least one test in which such a category is not represented in the training samples. This provides a way to test the classifier's performance on malicious flash files of a previously unknown category.

Error rates. Figure 1 contains the ROC curves displaying the accuracy of our classifier at various classification thresholds, using both manually selected and randomly selected malicious training samples. These ROC curves show the true positive rate as a function of the false positive rate, visualizing the trade-offs required to achieve a given accuracy.

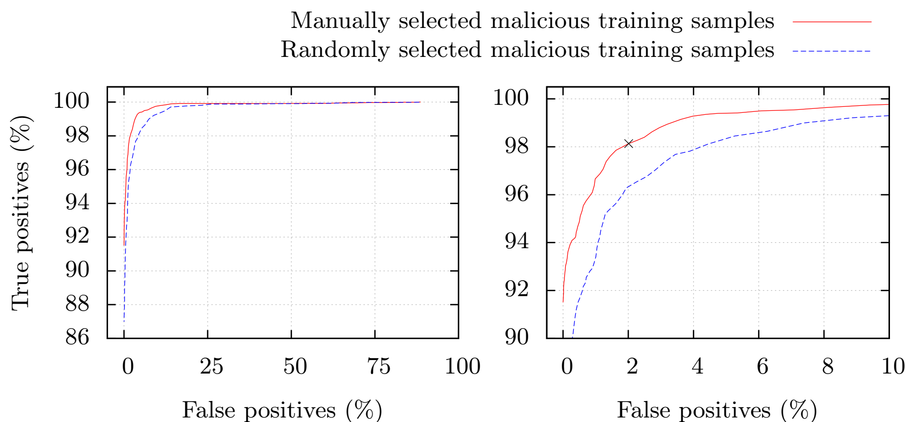

Fig. 1. ROC curves displaying FlashDetect's accuracy at various classification thresholds, when using manually or randomly selected training samples. The right-hand plot zooms in on the upper corner of the left-hand plot.

As shown in the plot, when using manually selected training samples, a false positive rate of 0% is reached at a true positive rate of around 91.5% (i.e. a false negative rate of around 8.5%). A true positive rate of more than 99.9% is

reached at a false positive rate of around 16%. However, a true positive rate of exactly 100% is only reached at a false positive rate of around 88.7%. Also note that an equilibrium between false positive and false negative rates is achieved at around 2%, at a classification threshold of 25 (marked with an 'x'). It is at this equilibrium that the classifier's accuracy is most balanced, and as such, this is the threshold we use in the next experiment, comparing our performance to that of commercial anti-virus products.

The plot also shows the performance of the classifier using randomly selected training samples, as described earlier. It is clear that the classifier is less accurate when using randomly selected samples than when using manually selected samples. However, the performance is not significantly worse, with an equilibrium between false positive and false negatives at around 3%, compared to 2% when using manually selected samples. Thus, we conclude that while manual sample selection increases the classifier's accuracy, it does not significantly bias towards malicious files of known categories.

Comparison with commercial anti-virus products. To support our claim that FlashDetect's performance is comparable with or better than that of commercial anti-virus (AV) products, we include a comparison of FlashDetect's efficacy with that of AV products. We used the VirusTotal [24] service to run 43 commercial and open-source AV products on our test sample set. The results of the five best-performing AV products, as determined by the false negative rate, are compared to FlashDetect's results, at a set classifier threshold of 25.

The first row of Table 3 lists the false negative rates for the five best-performing AV products, together with FlashDetect's rates. As shown, FlashDetect's average false negative rate is less than that of four out of the five best-performing AV products. It is also interesting to note that out of the 43 AV products we tested, 35 products had a false negative rate in excess of 65%, and 32 products had a false negative rate in excess of 90%.

The second row of Table 3 lists the false positive rates. Note that FlashDetect's false positive rate is worse than that of the top five AV products. This can be explained by the fact that these AV products probably use signature-based detection methods (confirmed for four out five). However, given that FlashDetect is able to reach a comparable false negative rate with a relatively low false positive rate, we conclude that FlashDetect's performance is certainly competitive with that of commercial AV products.

Table 3. FlashDetect's false negative and false positive rates at a classifier threshold of 25 compared to the five best-performing AV products.

	FlashDetect	AV1	AV2	AV3	AV4	AV5
False negatives	1.87%	0.97%	2.64%	2.73%	2.74%	2.81%
False positives	2.01%	0.42%	0.00%	0.27%	0.27%	0.27%

6 Limitations

Identifying the presence of FlashDetect. Malicious Flash files might try to identify the presence of FlashDetect. The Lightspark player is still a relatively young project, and thus, it still has a long way to go before its behavior perfectly matches that of the official Flash player. This provides malware writers with a number of ways to detect the presence of Lightspark, and as a result, FlashDetect. This limitation is inherent to the implementation of FlashDetect's dynamic analysis phase. However, we assume that as the Lightspark project matures it will become increasingly harder to differentiate Lightspark from the official Flash player.

Environment-identifying code circumvention. The method for handling environment-identifying code described in Section 2.6 is obviously not very robust. For example, the current method can easily be circumvented by using exclusive matching instead of inclusive matching. That is, matching against non-vulnerable version instead of vulnerable versions. Additionally, comparison instructions besides `ifeq` and `ifne` are not checked for environment-identification.

However, from our observations we conclude that, since most malicious Flash files currently found in the wild use inclusive environment-identification based on direct equality instructions, the current method is quite effective at enhancing detection rates. Nevertheless, in the future, a more robust method to handle environment-identifying code would be in order. Such a method could consist of a multi-execution virtual machine capable of simultaneously analyzing multiple code branches, such as described for JavaScript in [12].

Dependence of certain features on a measure of time. During the dynamic analysis phase, the Flash file is run for a limited amount of time. Therefore, the usefulness of the dynamic analysis phase inherently depends on the fact that, in general, malicious files will attempt exploitation as soon as possible. Additionally, certain dynamic features (e.g., *method-calls-per-second*) used for classification are inherently dependent on a measure of time.

The usefulness of time-dependent features and indeed, the dynamic analysis phase in general, could be reduced if malicious files were to delay the start of exploitation for a certain amount of time. Therefore, the dependence of certain features on a measure of time is an inherent limitation of our system.

However, launching an exploit as soon as possible is advantageous to malware authors as it increases the chances of the exploit being successful. Consequently, one can argue that maximizing the number of successful exploitations is more important to malware authors than evading detection. Indeed, the complete lack of obfuscation in some of the malicious samples we have observed indicates that some malware authors do not even bother to evade detection anymore.

Overall robustness of features. Some of the individual features used for classification may not be very robust. Examples are the time-dependent dynamic

features. However, the combination of all features being used together results in a robust system. Indeed, the features used detect a number of different types of behavior, such as obfuscation, JIT spraying, or environment-identification. Most of these types of behavior are detected by more than one feature, and they are often detected during both the dynamic and static analysis phase. This results in a robust system capable of detecting a wide range of low-level exploits.

7 Related work

7.1 Exploit techniques

Blazakis [2] discusses JIT spraying attacks against the ActionScript 3 virtual machine. Li [13] discusses the exploitation of the type confusion vulnerability CVE-2010-3654, while [9] discusses a very similar vulnerability CVE-2011-0609. An exploit using sophisticated obfuscation techniques that targets CVE-2007-0071 is dissected by Liu in [14].

7.2 Malware detection

FlashDetect's implementation is an evolution of OdoSwiff described by Ford *et al.* [6]. Additionally, [6] is one of only a very limited set of publications on the specific topic of malicious Flash file detection. Instead, most of the closest related research discusses malicious JavaScript detection approaches.

Cova *et al.* describe JSAND [3], a tool for analyzing JavaScript with an approach that is related to FlashDetect's approach, using classification based on a set of dynamic and static features. However, JSAND and FlashDetect use different features, due to the different nature of JavaScript and ActionScript.

Ratanaworabhan *et al.* [21] discuss Nozzle, a dynamic JavaScript analyzer that specifically focuses on detecting heap spraying code injection attacks. Nozzle's approach consists of interpreting individual objects on the heap as code and statically analyzing that code for maliciousness. This approach differs substantially from FlashDetect's approach, as FlashDetect's analysis is based on determining the general behavior of a Flash file through indicators such as the methods called by the file. Additionally, FlashDetect is not specifically focused on the detection of heap spraying exploits, but instead focuses on the more broader set of low-level exploits.

Zozzle [4] is a static JavaScript analyzer by Curtsinger *et al.* that also uses a naive Bayesian classifier to detect malicious files. However, the features used by Zozzle for classification are automatically extracted from the JavaScript's abstract syntax tree. In contrast, FlashDetect's static features are predefined, and it also uses predefined, dynamically extracted features.

The recent work on Rozzle by Kolbitsch *et al.* [12] describes an implementation for multi-execution in JavaScript. Their approach to multi-execution could be applied to Lightspark to enable the robust handling of Flash files using environment-identification.

There are several malware detection systems that use low-interaction or high-interaction honeyclients. Examples are HoneyMonkey [25], Capture-HPC [22], Moshcuk *et al.* [16,17], Provos *et al.* [20], and Monkey-Spider [7]. FlashDetect differs from such systems in that it does not automatically crawl websites. Instead, FlashDetect is designed to be used in conjunction with some other analyzer that feeds samples into FlashDetect for further analysis. Additionally, in contrast to high-interaction honeyclients, FlashDetect's analysis provides more insight into how an exploit works. High-interaction honeyclients on the other hand provide more insight into the effects of an exploit, something which FlashDetect does not currently do. However, in high-interaction honeypots, exploits must succeed for them to be detected, while this is not the case for FlashDetect.

8 Conclusion

We discussed several techniques commonly used by Flash malware. We have discussed how malware using ActionScript 3 often takes on a role of exploit facilitator, showing that a successful solution to detecting malicious Flash files is crucial. Subsequently, we have introduced FlashDetect, which uses a novel approach combining static and dynamic analysis to examine Flash files. FlashDetect's classification is based on a combination of predefined features. We have shown how these features, when used with a naive Bayesian classifier and a single vulnerability-specific filter, allow for high classification accuracy with a minimal amount of false negatives.

References

1. Adobe: Statistics: PC penetration, `http://www.adobe.com/products/flashplatformruntimes/statistics.edu.html`, accessed on 2012-06-15
2. Blazakis, D.: Interpreter exploitation: Pointer inference and JIT spraying (2010), `http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf`, accessed on 2012-06-15
3. Cova, M., Kruegel, C., Vigna, G.: Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In: Proceedings of the World Wide Web Conference (WWW). Raleigh, NC (April 2010)
4. Curtsinger, C., Livshits, B., Zorn, B., Seifert, C.: Zozzle: Low-overhead mostly static JavaScript malware detection. In: Proceedings of the Usenix Security Symposium (Aug 2011)
5. DoSWF.com: DoSWF - Flash encryption, `http://www.doswf.com/doswf`, accessed on 2012-06-15
6. Ford, S., Cova, M., Kruegel, C., Vigna, G.: Analyzing and detecting malicious flash advertisements. In: Proceedings of the 2009 Annual Computer Security Applications Conference. pp. 363–372. ACSAC '09, IEEE Computer Society, Washington, DC, USA (2009)
7. Ikinci, A., Holz, T., Freiling, F.: Monkey-spider: Detecting malicious websites with low-interaction honeyclients. In: In Proceedings of Sicherheit, Schutz und Zuverlässigkeit (2008)

8. JavaScript-Source.com: JavaScript obfuscator, http://javascript-source.com, accessed on 2012-06-15
9. Joly, N.: Technical Analysis and Advanced Exploitation of Adobe Flash 0-Day (CVE-2011-0609) (2011), http://www.vupen.com/blog/20110326.Technical_Analysis_and_Win7_Exploitation_Adobe_Flash_0Day_CVE-2011-0609.php, accessed on 2012-06-15
10. Keizer, G.: Attackers exploit latest Flash bug on large scale, says researcher, http://www.computerworld.com/s/article/9217758/Attackers_exploit_latest_Flash_bug_on_large_scale_says_researcher, accessed on 2012-06-15
11. Kindi: secureSWF, http://www.kindi.com, accessed on 2012-06-15
12. Kolbitsch, C., Livshits, B., Zorn, B., Seifert, C.: Rozzle: De-cloaking internet malware. In: IEEE Symposium on Security and Privacy (May 2012)
13. Li, H.: Understanding and Exploiting Flash ActionScript Vulnerabilities. CanSecWest 2011 (2011), http://www.fortiguard.com/sites/default/files/CanSecWest2011_Flash_ActionScript.pdf, accessed on 2012-06-15
14. Liu, B.: Flash mob episode II: Attack of the clones (2009), http://blog.fortinet.com/flash-mob-episode-ii-attack-of-the-clones/, accessed on 2012-06-15
15. MITRE Corporation: Common Vulnerabilities and Exposures (CVE), http://cve.mitre.org, accessed on 2012-06-15
16. Moshchuk, A., Bragin, T., Deville, D., Gribble, S.D., Levy, H.M.: Spyproxy: execution-based detection of malicious web content. In: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium. pp. 3:1–3:16. SS'07, USENIX Association, Berkeley, CA, USA (2007), http://dl.acm.org/citation.cfm?id=1362903.1362906
17. Moshchuk, E., Bragin, T., Gribble, S.D., Levy, H.M.: A crawler-based study of spyware on the web (2006)
18. Paget, F.: McAfee Blog: Surrounded by Malicious PDFs, http://blogs.mcafee.com/mcafee-labs/surrounded-by-malicious-pdfs, accessed on 2012-06-15
19. Pignotti, Alessandro, e.a.: Lightspark flash player (2008), http://lightspark.github.com, accessed on 2012-06-15
20. Provos, N., Mavrommatis, P., Rajab, M.A., Monrose, F.: All your iframes point to us. In: Proceedings of the 17th conference on Security symposium. pp. 1–15. SS'08, USENIX Association, Berkeley, CA, USA (2008), http://dl.acm.org/citation.cfm?id=1496711.1496712
21. Ratanaworabhan, P., Livshits, B., Zorn, B.: Nozzle: A defense against heap-spraying code injection attacks. In: Proceedings of the Usenix Security Symposium (Aug 2009)
22. The HoneyNet Project: CaptureHPC, https://projects.honeynet.org/capture-hpc, accessed on 2012-06-15
23. Tung, L.: Flash exploits increase 40 fold in 2011, http://www.cso.com.au/article/403805/flash_exploits_increase_40_fold_2011, accessed on 2012-06-15
24. VirusTotal: VirusTotal service, https://www.virustotal.com, accessed on 2012-06-15
25. Wang, Y.M., Beck, D., Jiang, X., Roussev, R.: Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In: IN NDSS (2006)