SWF and the Malware Tragedy
Hide and Seek in a Flash

Ben Fuhrmannek[1] and fukami[2]

[1] bef@erlangen.ccc.de, http://pentaphase.de/
[2] SektionEins GmbH, fukami@sektioneins.de, http://sektioneins.de/

December 13, 2008

Abstract The analogy of a children's game of hide and seek describes the constant back and forth of attacks involving Flash on one hand and increasingly sophisticated analysis methods for static code analysis on the other hand. One point in the game utilises heuristic methods to recognise malware, followed closely by simple obfuscation methods to be safely concealed once more.

Introduction

Would you like to play a game of "hide and seek"? Imagine yourself to be just seven or eight years old, playing in the nearby park with your neighbours. First round - you are among the hiding group, scatter out and find a suitable spot behind a huge tree. Soon enough the tree turns out to be a rather obvious choice, too easy to be found. Second round - you are a seeker with a skilled eye for the very hideouts you evaluated a few moments ago for yourself. Since the hiding group knows where you hid in the first place, they try to come up with different places and strategies in order to be concealed for yet another round. Wood is being moved around as diversion or to build new hiding locations. One group disguises as the other to get information on where the others might be hiding or seeking. Successive rounds reveal increasingly complex strategies where each group learns from the other and incorporates previous tactics into their own.

Just like children, virus writers and the antivirus industry have been playing this game for a long time [3]. We can assume, that attackers try to conceal both the existence of their virus code itself and the functionality built into the code. Naturally our seekers in this game try to recognise potential threats and analyse suspicious code.

The major strategy to detect a virus without execution is a signature dictionary: Characteristics of previously classified code form a signature to be used as search pattern. Unfortunately, new or changed code requires a new signature to be detected this way. This weakness is addressed by polymorphic or metamorphic code (see [4]). In order to be still able to detect this constantly changing code, statistical classification comes to mind.

[3] according to [5] the first virus was released in the early 1970s

The same game of hide and seek can be applied to Adobe Flash as well. The seeker's job is to find an attacker's payload before execution. Naturally attackers want to delay the recognition of their payload as long as possible, but at least up to the point of execution. Common attacks involving Flash and static analysis using equivalents to a signature dictionary are described in detail in [2].

Assuming a dynamic payload generation so that static patterns are not applicable as recognition method, it will be shown, how a form of statistical classification can be adapted and how to overcome the heuristics. But first of all, let's have a look around the Flash playground.

Flash Playground

Most commonly an attacker may try to exploit the Flash player, analysis tools, the client (e.g. by stealing credentials) or even try to use a client's CPU time for their own purposes. Although some of these goals may be achieved purely by attacking the format parser of the player or tools, e.g. by fuzzing the SWF format, a parser crash would most likely arise suspicion. Therefore a payload is assumed to be present for the attack.

SWF as defined in [3] is a container format combining arbitrary resources into a single file. Common resources would be images, fonts, audio or video data and some application logic as compiled bytecode. The Flash player contains two entirely separate virtual machines - AVM1 and AVM2[4]. The byte loader feature offered by AVM2, which aims to load data from a byte array during runtime, is of particular interest. This way, encoded, encrypted or otherwise obscured data can be interpreted as SWF without being recognisable as SWF before execution. Starting from Flash version 10 fast memory operations on byte arrays[5] help preparing data to be loaded by the byte loader very efficiently. One example would be an OGG decoder implemented with fast memory operations (see [8]).

Then there is the possibility to program triggers. Arbitrary information available to the Flash runtime environment can be used to trigger the execution of a payload in order to prevent runtime analysis tools detecting an unexpected behaviour. Data such as the Flash player version, browser version, JavaScript interpreter version, IP address, the current date and time and SWF source URL are known to have been used as triggers to effect only specific targets. This data could for example be used to form a crypto key for decoding a payload.

Statistical Classification

For the analysis of non-static code with static function range, a classification method based on n-grams was suggested (see [1]). The front row application for n-grams is language detection of written text. The occurrence of every N successive

[4] AVM1 up to version 8, AVM1 and AVM2 starting from version 9

[5] Efficient fast memory operations[6] have been introduced along with the compiler toolkit Alchemy[7].

characters (including whitespace) of a text is counted and then compared relative to a reference count of known language.

This classification method can be applied to Flash as well. Instead of n characters of a text, a sequence of n opcodes can serve as a basis. For example, a 2-gram profile of the opcode sequence "push, push, add, push, add, return" contains the following pairs of opcode sequences and corresponding counts:

opcodes	count
push, add	2
push, push	1
add, push	1
add, return	1

The distance of two profiles is calculated as simplified distance as suggested in [1]:

$$\delta(SP_r, SP_p) = \sum_{n \in \{op(SP_r) \cap op(SP_p)\}} \left(\frac{2(f_r(n) - f_p(n))}{f_r(n) + f_p(n)} \right)^2$$

where $f_1(n)$ and $f_2(n)$ are the normalised counts of profiles, $op(profile)$ yields all opcodes of a profile and SP_r and SP_p are n-gram profiles.

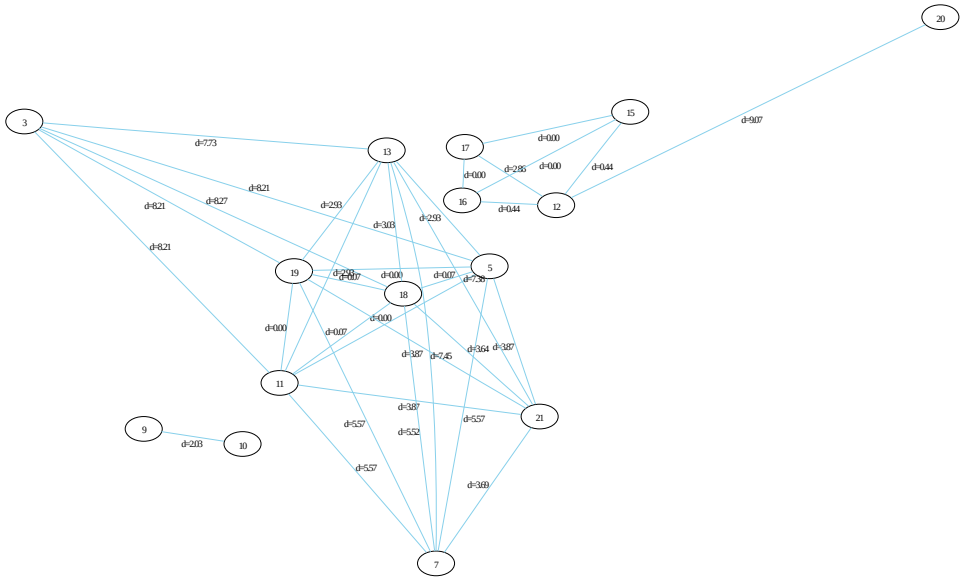

Figure 1. distance graph of n-gram profiles of AVM2 bytecode

Figure 1 shows a graph representation of several SWF9/10 files[6] and their distance based upon the n-gram analysis. Edges connecting far nodes have been eliminated by distance threshold. Three clusters become apparent: {9,10}, {17,16,12,15,20} and {3,13,19,11,18,5,21,7}. Clustering is an expression of similarity between the SWF's bytecode. N-gram profiles contain characteristics of the compiler or IDE, std. libraries, the code's functionality and the code's author(s), each with different intensity. A proof of concept implementation of the Flash adaptation of the algorithm is part of erlswf (see [9]).

Obfuscation and Data Hiding

Initially an attacker would like to conceal the very existence of a payload, however, if found suspicious, it should be as hard as possible to analyse the payload's application logic. These two goals work well together. In order to render the statistical classification ineffective, the payload could either imitate suspected reference characteristics or use as little code as possible. By hiding most of the code, thus leaving only a decoding function exposed to the classification test, both strategies - concealment and obstructing the analysis - can be accommodated.

The following haXe[7] code is a simple example for decoding, loading and executing a payload embedded in SWF using the AVM2 byte loader:

```
class Test {
  static function main() {
    var ldr : Dynamic = new flash.display.Loader();
    var ba : flash.utils.ByteArray = new flash.utils.ByteArray();
    for (i in Data.data) {
      ba.writeByte(i-1); // secret de-obfuscation algorithm: i - 1
    }
    ldr.loadBytes(ba);
    flash.Lib.current.addChild(ldr);
  }
}
```

In a similar way, a payload could have been loaded from an external source, e.g. a picture from the internet, decoded, possibly decrypted, then loaded and executed.

Conclusion

Statistical classification can be effectively applied to identify bytecode similar to a reference profile only up to the point, where characteristics of the reference profile can be approximated by a payload or too few data exists to analyse

[6] files were chosen to contain a suitable amount of AVM2 bytecode
[7] see [10]

bytecode effectively. In the end it always breaks down to a matter of how much effort each side is willing to put into either obfuscation or analysis respectively. Malware seen in real-life at the moment hardly makes any effort to be not easily detected by heuristic methods. This is expected to change in the future, as well as more use of the byte loader and fast memory operations.

Also consider the possibility of one side of the hide and seek game to be tricked into making the next move, just to be observed by the opponent and learn from the reaction. What if you don't even know whether your are currently playing the game or not? You may be hiding accidentally right now and don't even realise it. Every new pattern recognition algorithm, coding algorithm or obfuscation idea will be taken into account and can most likely be adapted to serve as the next round of the game. So the question remains: Would you like to play a game of "hide and seek"?[8]

References

1. J. Filipe et al. (Eds.): Supprting the Cybercrime Investigation Process: Effective Discrimination of Source Code Authors Based on Byte-Level Information - ICETE 2005, CCIS 3, pp. 163173, Springer-Verlag (2007)
2. fukami and Ben Fuhrmannek: SWF and the Malware Tragedy: Detecting Malicious Adobe Flash Files, https://www.flashsec.org/mediawiki/images/5/57/SWF_and_the_Malware_Tragedy.pdf, March 9, 2008
3. Adobe Inc., SWF Format Specification version 10, Specification, http://www.adobe.com/devnet/swf/
4. Wikipedia: The Free Encyclopedia: Polymorphic Code, http://en.wikipedia.org/wiki/Polymorphic_code
5. Wikipedia: The Free Encyclopedia: Computer Virus, http://en.wikipedia.org/wiki/Computer_virus
6. Scott Petersen, FlaCC, Presentation Slides, http://llvm.org/devmtg/2008-08/Petersen_FlashCCompiler.pdf, August 1, 2008
7. Adobe Alchemy, Tool, http://labs.adobe.com/technologies/alchemy/
8. Adobe Alchemy, Code Example, http://labs.adobe.com/wiki/index.php/Alchemy:Libraries
9. Ben Fuhrmannek, erlswf: Toolkit for disassembling SWF up to version 10. Tool, http://code.google.com/p/erlswf/
10. Motion-Twin, haXe, Tool, http://haxe.org/
11. Ben Fuhrmannek and fukami: SWF and the Malware Tragedy, 25C3 Presentation, http://events.ccc.de/congress/2008/Fahrplan/events/2596.en.html, December 2008

License

[8] For more elaborate answers to this question see [11].