# *k*-Depth Mimicry Attack to Secretly Embed Shellcode into PDF Files

Jaewoo Park and Hyoungshick Kim

Department of Software, Sungkyunkwan University
2066 Seobu-ro, Suwon, Republic of Korea
{bluereaper,hyoung}@skku.edu

**Abstract.** This paper revisits the shellcode embedding problem for PDF files. We found that a popularly used shellcode embedding technique called *reverse mimicry* attack has not been shown to be effective against well-trained state-of-the-art detectors. To overcome the limitation of the *reverse mimicry* method against existing shellcode detectors, we extend the idea of *reverse mimicry* attack to a more generalized one by applying the $k$-depth mimicry method to PDF files. We implement a proof-of-concept tool for the $k$-depth mimicry attack and show its feasibility by generating shellcode-embedded PDF files to evade the best known shellcode detector (`PDFrate`) with three classifiers. The experimental results show that all tested classifiers failed to effectively detect the shellcode embedded by the $k$-depth mimicry method when $k \geq 20$.

**Keywords:** Security, Malware, PDF, Shellcode, Mimicry attack

## 1 Introduction

Portable Document Format (PDF) based on "ISO 32000-1:2008 14.8" [7], originally created by Adobe [5], has become the de-facto standard for document files in the web environment, which accounts for over 75% of all online document files [8]. Naturally, the popularity of PDF files made them attractive targets [13] for cyber criminals who want to distribute malware through online document files. Many previous studies have tried to develop efficient methods to embed shellcode into a PDF document and defense mechanisms [6, 9, 12] to detect the embedded shellcode in PDF documents.

Mimicry attack [17] is a widely used general concept to cloak an attacker's intrusion to avoid detection by security solutions. Mimicry attack has also been applied to secretly embed shellcode into a document file (e.g., PDF and Microsoft Office files) against shellcode detection techniques by blending shellcode with *normal* document objects. Maiorca [9] particularly proposed an automatic shellcode embedding technique called *reverse mimicry* by hiding a given malicious code as an object linked to the parent object through an *indirect reference*. This technique might be effective against the detectors based on PDF structure analysis (e.g., `PJScan` [1]). However, the effectiveness of *reverse mimicry* is still questionable against the-state-of-art detection technologies (e.g., `Malware`

`Slayer` [2] and `PDFrate` [3]) that were developed based on machine learning to statistically distinguish the patterns of malicious PDF documents from those of benign PDF documents. In this paper, we show that the *reverse mimicry* method is not effective against `PDFrate`. Instead, we extend the idea of conventional *reverse mimicry* attack into a more generalized attack method against PDF files called $k$-depth mimicry attack by using $k$ *indirect references* repeatedly. Through the $k$-depth mimicry attack method, we show the clear limitation of existing PDF shellcode detection tools. Our key contributions are as follows:

- **Design of a new PDF shellcode embedding technique.** We developed a new PDF shellcode embedding technique called $k$-depth mimicry attack, which can be used to secretly embed a JavaScript code into a PDF file.
- **Evaluation of $k$-depth mimicry attack against PDF shellcode detection tools.** We showed the effectiveness of the $k$-depth mimicry attack through experiments with popularly used PDF shellcode detection tool (`PDFrate`). Our experimental results demonstrate that the $k$-depth mimicry attack is significantly more effective in hiding the embedded shellcode in a PDF file than the conventional PDF shellcode embedding technique such as "reverse mimicry" [9].

The rest of this paper is organized as follows. In Section 2, we provide the background information about PDF files to understand the shellcode embedding attack and defense techniques. In Section 3, we present $k$-depth mimicry attack. In Section 4, we evaluate the effectiveness of $k$-depth mimicry attack compared with the performance of traditional mimicry attack. Finally, we conclude in Section 5.

## 2 Structure of PDF

In this section, we provide the basic information about the structure of PDF file. In general, a PDF file consists of a sequence of *objects* (e.g., fonts, pages and images) representing components of a document [4]. The description of PDF objects is explained in the ISO 32000-1 standard [7].

### 2.1 Object, categorized by Types

Basically, objects in a PDF document are categorized into the following types:

- `Array`. This object is an elements list, enclosed in square brackets (i.e., `[`$\sim$`]`). This object is generally used to store the reference information of multiple objects.
- `Boolean`. This object represents a logical value (i.e., `TRUE` or `FALSE`).
- `Dictionary`. This object contains one or more entries that are enclosed in double angle brackets (i.e., `<<` $\sim$ `>>`). This is a collection of key and value pairs. Each key is always a name object while the value may be any other type of object, including another dictionary or even null. The dictionary object is one of the most common objects in PDF files.

- `Name`. This object is a sequence of characters starting with a slash character "/" for a fixed value set. This is mainly used for an entry's attribute set such as "`/Name`", "`/Type`" and "`/Filter`".
- `Null`. The string `null` represents an empty object.
- `Numeric`. This object represents an integer or a real number.
- `Stream`. This object is a sequence of bytes to store large blobs of data that are in some other standardized format, such as XML grammars, font files, and image data. This starts with a specific string of `0x73747265616D` to indicate the start position of a stream object, and ends with another specific string of `0x656E6473747265616D` to indicate the end position of the stream object.
- `String`. This object represents a text string, which is enclosed in parentheses or angle brackets (i.e., `(` $\sim$ `)`, `<` $\sim$ `>`).

In general, each object could have attributes called `entry` to specify the properties of the object. There are many different types of entries. `OpenAction` and `Annots` are popularly used. `OpenAction` entry is used to define an action to be taken after opening a PDF file. `Annots` is used to indirectly refer to another object such as text note, link, or embedded file. Naturally, those entries could be misused for injecting shellcode.

Objects could be labeled so that they can be referred to by other objects. A labeled object is called an *indirect object*.

### 2.2 File Structure

In general, a PDF file consists of four primary sections: header, body, Cross-reference table and Trailer. The header section contains the basic information (e.g., format version) about PDF file. The body section consists of a set of objects. The cross-reference (also known as 'Xref') table section is a large dictionary for the indexes by which all of the indirect objects, in the PDF file, are located. The trailer section contains the offset and object number information of `root`, the Xref's starting point offset, etc. When these irremovable parts are complete, End–of–File marker "%%EOF" is used to indicate the end of a file.

### 2.3 Document Structure

In a PDF file, all objects are organized into a tree structure. This stems from the primary objects (i.e., `root` or `catalog`). A PDF document consists of objects contained in the body section of the PDF file. Each page of the document is represented by a `page` object, which is a dictionary that includes references to the page's contents. Page objects are connected together and form a page tree, which is declared with an indirect reference in the document `catalog`. Each `page` object has its children objects called `Kids` that specifies all the children objects directly accessible from the current node. A PDF viewer application generally parses this tree structure of objects and renders the objects visible to the user.

### 2.4 Common shellcode types

*Shellcode* is a sequence of machine language instructions that can be injected into a running application. For PDF files, three different file types have been popularly used as shellcode that can be embedded inside a PDF file: (1) an executable file, (2) another PDF file and a `JavaScript` code.

## 3 *k*-depth Mimicry Attack

In this paper, we introduce a new shellcode injection method for PDF files called "*k*-depth mimicry" attack. We designed this sophisticated attack to secretly hide shellcode against existing shellcode detection techniques.

In a shellcode injection method, an exploit is crafted to fool the targeted application (e.g., Microsoft Office and Adobe PDF reader) into executing a malicious code, which is hidden within a document (e.g., office file) as shellcode. Several defense solutions [11,14,18] have been developed to prevent the shellcode injection by analyzing the difference between malware and normal applications because this technique was very popularly used. However, many advanced shellcode injection techniques have also been proposed to make it more difficult to detect the presence of shellcode [10]. In a PDF file, the *indirect reference* could be used to avoid shellcode detection methods, by scattering the embedded shellcode across objects [15]. Figure 1 shows how the reverse mimicry attack works with an indirect reference to a JavaScript code within the PDF document.

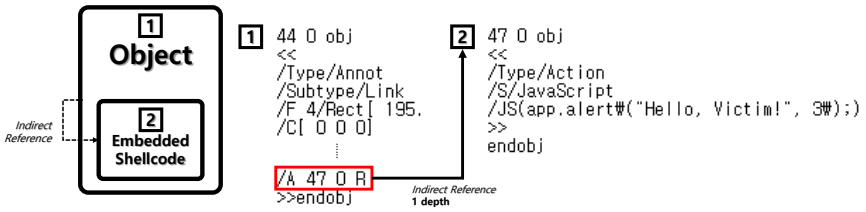


**Figure 1.** Overview of reverse mimicry.

As depicted in Figure 1, the reverse mimicry method modifies an object to have an indirect reference link to another object. The indirectly linked object is the carrier of the embedded shellcode. In the example of Figure 1, the reverse mimicry method uses a seemingly harmless object as an intermediate, hiding the malicious shellcode underneath that object with an indirect reference to the "`47 0 obj`" object. When the "`44 0 obj`" object is rendered to show its contents, its indirect reference object "`47 0 obj`" would be sequentially launched through the object "`44 0 obj`".

We extend the conventional reverse mimicry attack into a more generalized one by hiding shellcode with an indirect object with depth $k$. We call this technique "$k$-depth mimicry attack". Here, *depth* is defined as the number of

indirect references from a referencing object to a referenced object. In the reverse mimicry method, one indirection reference is only used to hide the malicious shellcode, and thus, reverse mimicry can be regarded as the "1-depth" mimicry.

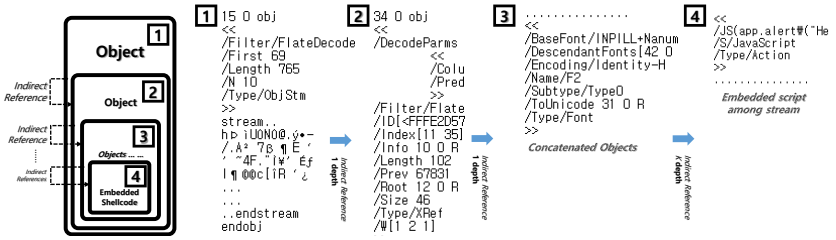


**Figure 2.** Overview of $k$-depth mimicry.

Figure 2 shows how the $k$-depth mimicry attack works. As depicted in the Figure 2, basically, each object has an indirect reference to the next object. In this chain of indirect references, the last object is the carrier with shellcode. We note that the main idea of the $k$-depth mimicry attack is to use multiple layers of indirect references to make it harder to extract the injected shellcode.

In the example of Figure 2, "`15 0 obj`" object is indirectly linked to "`34 0 obj`" object, "`34 0 obj`" holds the indirect reference to another object, and so on. After $k$ times of such indirect references among objects (e.g., $k$=4 in Figure 2), the last object with the JavaScript shellcode can be reached. This concatenation is launched when the PDF document is opened and its contents are rendered. The $k$-depth mimicry is similar to the reverse mimicry method except that $k$ subsequent indirect references are used. However, we found that it is much harder to detect the presence of shellcode in practice when a reasonable number $k$ of indirect references is used for hiding the shellcode.

In the following section, we will show that the $k$-depth mimicry attack is more effective than the reverse mimicry attack against even machine-learning based detectors through several experiments.

## 4 Experiments

For experiments, we created a normal PDF file (benign) using the $k$-depth mimicry technique to embed a JavaScript code (as shellcode) into the PDF file and then analyzed the file with the PDF shellcode detector called `PDFrate` [3] that is one of the most popular tools for malicious PDF detectors. `PDFrate` has achieved a high detection rate and a low false positive rate. Moreover, its performance has improved over time because it is publicly available and its training datasets are continuously updated by online users.

To find the optimal $k$ for the $k$-depth mimicry method, we created several test PDF files with varying depth $k$ and analyzed their classification scores

in PDFrate. We used the three different classifiers deployed by PDFrate (i.e., the classifiers trained on the Contagio, George Mason University (GMU), and PDFrate community (Community) datasets [16]). Figure 3 shows how these scores are changed with $k$.

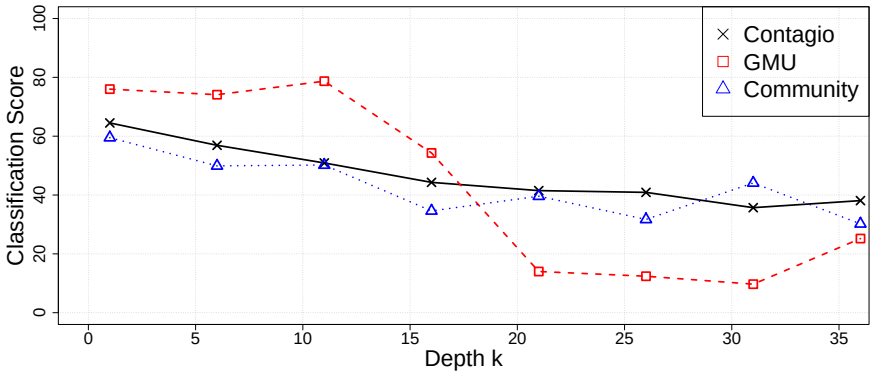


**Figure 3.** Changes in the classification score with depth $k$.

From this figure, as expected, the performance of all classification scores overall reduced as $k$ decreased. The classification scores of GMU dramatically decreased from $k = 10$ to 20 while the score curves of the other classifiers commonly had a gentle slope. Based on those experimental results, we recommend using a reasonably large $k \geq 20$ for the $k$-depth mimicry method to significantly reduce the classification scores of PDF files' maliciousness.

To show the effectiveness of the $k$-depth mimicry attack on PDF file, we created the PDF file with the depth of 21 and compared its maliciousness on PDFrate with the PDF file created by the reverse mimicry technique. Table 1 shows the experimental results.

**Table 1.** Analysis results of PDFrate with Benign PDF, PDF by reverse mimicry, PDF by $k$-depth mimicry (Each percentage in parenthesis indicates a classification score of the tested PDF file's maliciousness).

| Classifier | Benign PDF | PDF by reverse mimicry | PDF by $k$-depth mimicry |
|---|---|---|---|
| Contagio | – (20.4%) | Detected (50.9%) | **Evaded** (41.5%) |
| GMU | – ( 5.6%) | Detected (78.7%) | **Evaded** (14.0%) |
| Community | – (27.6%) | Detected (50.2%) | **Evaded** (39.6%) |

In Table 1, we can see that all those classifiers failed to successfully detect the shellcode embedded by the $k$-depth mimicry technique. Even though the classification scores returned by PDFrate for the PDF file by the $k$-depth mimicry technique were overall higher than those scores for the benign PDF file with no JavaScript code, the $k$-depth mimicry attacks significantly reduce the classification

scores of PDFrate from the mean of about 59.9% for the reverse mimicry technique to the mean of about 31.7%. Those experimental results demonstrate that the $k$-depth mimicry technique is practically effective for hiding shellcode against existing PDF shellcode detectors such as PDFrate.

To make matters worse, even for the latest version of Adobe Reader DC (v15.023.20053), the PDF document containing the JavaScript embedded by the $k$-depth mimicry method was successfully opened with no security warning whereas the PDF document was not opened when the reverse mimicry method was used to embed the same JavaScript code.

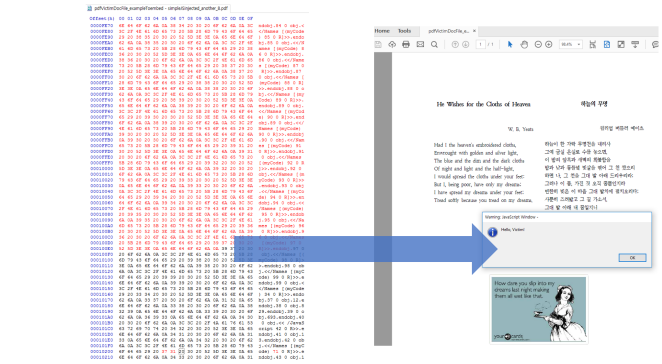


**Figure 4.** Example of the embedded shellcode using the $k$-depth mimicry technique (with 21–depth).

## 5 Conclusion

We presented a novel shellcode embedding technique for PDF documents to successfully bypass PDF shellcode detectors. Our experimental results showed the proposed technique can be used to easily create a PDF file with shellcode against the state-of-the-art detectors such as `PDFrate`. However, our current results are not enough to generalize our observations because a single PDF file was tested. Therefore, as an extension to this paper, we plan to conduct intensive experiments with a large sample of PDF files.

In future work, we plan to develop defense mechanisms so as to make the $k$-depth mimicry technique ineffective. It would also be interesting to evaluate the performance of the defense mechanisms on real malicious PDF files.

**Acknowledgement.** This work was supported in part by the NRF Korea (No. 2014R1A1A1003707), the ITRC (IITP-2016-R0992-16-1006), and the MSIP/IITP (No. R0166-15-1041, R-20160222-002755). The first author's research was mainly funded by the MSIP, under the "Employment Contract based Master's Degree Program for Information Security" (H2101-16-1001) supervised by KISA. The contents of this article do not necessarily express the views of KISA.

## References


1. PJScan (2013), https://sourceforge.net/p/pjscan/home/Home
2. Malware Slayer (2014), https://pralab.diee.unica.it/en/Slayer
3. PDFrate (2016), https://csmutz.com/pdfrate
4. Adobe Systems Incorporated: PDF reference–adobe portable document format (2006), https://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/pdf_reference_1-7.pdf
5. Adobe Systems Incorporated: What is PDF? (2016), https://acrobat.adobe.com/kr/ko/why-adobe/about-adobe-pdf.html
6. Fratantonio, Y., Kruegel, C., Vigna, G.: Shellzer: a tool for the dynamic analysis of malicious shellcode. In: Proceedings of International Workshop on Recent Advances in Intrusion Detection (2011)
7. International Organization for Standardization: PDF (portable document format), version 1.7, base level (iso 32000-1:2008) (2008), http://www.digitalpreservation.gov/formats/fdd/fdd000277.shtml
8. Johnson, D.: PDF still dominates electronic documents online (2015), http://duff-johnson.com/2015/10/07/pdf-still-dominates-electronic-documents-online
9. Maiorca, D., Corona, I., Giacinto, G.: Looking at the Bag is Not Enough to Find the Bomb: An Evasion of Structural Methods for Malicious PDF Files Detection. In: Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (2013)
10. Mason, J., Small, S., Monrose, F., MacManus, G.: English Shellcode. In: Proceedings of the 16th ACM Symposium on Information, Computer and Communications Security (2009)
11. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Emulation-based Detection of Non-self-contained Polymorphic Shellcode. In: Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection (2007)
12. Schmitt, F., Gassen, J., Gerhards-Padilla, E.: PDF Scrutinizer: Detecting JavaScript-based Attacks in PDF Documents. In: Proceedings of the 10th Annual International Conference on Privacy, Security and Trust (2012)
13. Symantec: ISTR: Internet security threat report. In: Trend Report, VOLUME 21 (2016)
14. Toth, T., Kruegel, C.: Accurate Buffer Overflow Detection via Abstract Payload Execution. In: Proceedings of the 5th International Conference on Recent Advances in Intrusion Detection (2002)
15. Tzermias, Z., Sykiotakis, G., Polychronakis, M., Markatos, E.P.: Combining static and dynamic analysis for the detection of malicious documents. In: Proceedings of the 4th European Workshop on System Security (2011)
16. Šrndic, N., Laskov, P.: Practical Evasion of a Learning-Based Classifier: A Case Study. In: Proceedings of the IEEE Symposium on Security and Privacy (2014)
17. Wagner, D., Soto, P.: Mimicry Attacks on Host-based Intrusion Detection Systems. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (2002)
18. Zhang, Q., Reeves, D.S., Ning, P., Iyer, S.P.: Analyzing Network Traffic to Detect Self-decrypting Exploit Code. In: Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (2007)