O-checker: Detection of Malicious Documents through Deviation from File Format Specifications

Yuhei Otsubo
National Police Agency, Japan
National center of Incident readiness and Strategy for Cybersecurity, Japan
Institute of Information Security, dgs157101@iisec.ac.jp

Mamoru Mimura
JMSDF Command and Staff College
Institute of Information Security

Hidehiko Tanaka
Institute of Information Security

Abstract

Documents containing executable files are often used in targeted email attacks in Japan. We examine various document formats (Rich Text Format, Compound File Binary and Portable Document Format) for files used in targeted attacks from 2009 to 2012 in Japan. Almost all the examined document files contain executable files that ignore the document file format specifications. Therefore, we focus on deviations from file format specifications and examine stealth techniques for hiding executable files. We classify eight anomalous structures and create a tool named "o-checker" to detect them. O-checker detects 96.1% of the malicious files used in targeted email attacks in 2013 and 2014. There are far fewer stealth techniques than vulnerabilities of document processors. Additionally, document file formats are more stable than document processors themselves. Accordingly, we assert that o-checker can continue detecting malware with a high detection rate for long periods.

1 Introduction

The threat of targeted email attacks has increased in Japan in recent years. Many Japanese organizations have received targeted emails, some of which resulted in leaked confidential information. There were many such incidents in 2015.

In a targeted email attack, an email requests that the recipient open an attached file or click on a hyperlink in the email body. Over 60% of the attachment files in targeted email attacks occurring in 2014 were document files [21]. When the attachment file is an executable file, we can avoid malware by checking the file extension. However, when the attachment file is a document file, we cannot determine risk from the file extension alone.

Malicious document files used in targeted email attacks often contain an executable file embedded within a decoy document file. Figure 1 shows a typical structure and execution process of a malicious document file.

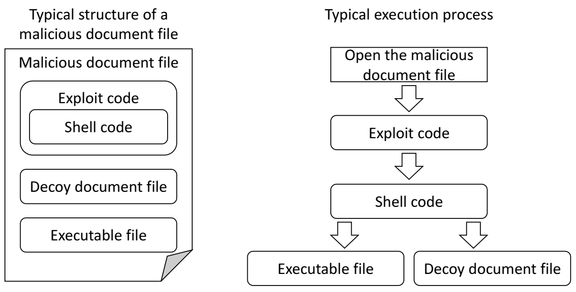


Figure 1: Structure and execution process of a malicious document file.

The left-hand side of Fig. 1 shows a typical structure for a malicious document file. A malicious document file mainly consists of four parts: exploit code, shell code, an executable file, and a decoy document file. Exploit code is a program designed to exploit a document processor vulnerability. Shell code is a program designed to create an executable file and a decoy document file, and to launch the executable file.

The right-hand side of Fig. 1 shows a typical execution process of a malicious document file. The exploit code is executed when a malicious document file is opened, leading to execution of the shell code.

Generally, document processors process exploit code, which targets vulnerabilities in the document processor itself, but the executable file and decoy document file are rarely processed by document processors. If these files are processed, the displayed content is garbled, or the software leads to strange behavior. The executable file is thus present in the stored content, but not displayed to the user. This mismatch between displayed and stored content is the key detection method in o-checker.

O-checker is a tool for detecting malicious document files. It does not analyze exploit code, shell code, executable files, or decoy documents. Instead, o-checker

Table 1: Summary of Specimens 1.

Type	Ext.	Num.	Avg. Size (KB)
RTF	rtf	98	266.5
CFB	doc	36	252.2
	xls	49	180.4
	jtd/jtdc	17	268.5
PDF	pdf	164	351.2
Total	-	364	291.8

Table 2: Vulnerabilities targeted in 2012 by Specimens 1.

Vulnerability	Num.	Rate
MS09-67	8 / 130	6.2%
MS10-087	31 / 130	23.4%
MS11-021	2 / 130	1.5%
MS12-027	51 / 130	39.2%
APSB09-04	2 / 130	1.5%
APSB10-02	1 / 130	0.8%
APSB10-07	3 / 130	2.3%
APSB10-21	8 / 130	6.2%
APSB11-07	2 / 130	1.5%
APSB11-08	10 / 130	7.7%
APSB11-30	1 / 130	0.8%
APSB12-03	1 / 130	0.8%
APSB12-18	10 / 130	7.7%
APSB12-22	5 / 130	3.8%
None	2 / 130	1.5%

examines deviation from file format specifications. O-checker runs quickly and detects malicious document files containing executable files at a fairly high rate. This paper describes the detection methods of o-checker and evaluates its effectiveness.

2 Structure of document files

We analyzed 364 document files gathered from various Japanese organizations. These specimens were used in targeted email attacks from 2009 to 2012, and each contained an executable file. We call these 364 specimens Specimens 1.

Table 1 shows a summary of Specimens 1. The specimens consist of three file formats: Rich Text Format (RTF; rtf files), Compound File Binary (CFB; doc, xls, or jtd/jtdc files), and Portable Document Format (PDF; pdf files).

Among Specimens 1, 130 document files were used in 2012. Table 2 shows the vulnerabilities targeted by these 130 files. The most frequently targeted vulnerability was MS12-027. The total in the Rate column (rightmost column of Table 2) is more than 100% because some of the specimens targeted multiple vulnerabilities.

Table 3: Rate of each anomalous structure.

Type	Anomalous structures	Num.	Rate
RTF	AS1	97 / 98	99.0%
CFB	AS2	79 / 102	77.5%
	AS3	92 / 102	90.2%
	AS4	99 / 102	97.1%
	AS5	98 / 102	96.1%
	AS2, AS3, AS4, or AS5	100 / 102	98.0%
PDF	AS6	81 / 164	49.4%
	AS7	72 / 164	43.9%
	AS8	104 / 164	63.4%
	AS6, AS7, or AS8	163 / 164	99.4%

Although there are various vulnerabilities of document processors [7, 5, 6, 3, 4], the stealth techniques for hiding executable files were simple. In the case of RTF or CFB files, almost all the positions of the contained executable files were the end of the file. In the case of PDF files, almost all the positions were the end of the file or the inside of a stream. On the other hand, the contained executable files were usually encrypted to avoid detection based on pattern matching. Therefore, we examined the specimens for deviations from file format specifications. The examination revealed eight anomalous structures (AS).

- RTF: AS1
- CFB: AS2, AS3, AS4, and AS5
- PDF: AS6, AS7, and AS8

Table 3 shows that we could classify almost all the specimens according to the eight anomalous structures.

The following sections describe the eight anomalous structures.

2.1 RTF

2.1.1 Structure

This section describes an overview of the RTF specification.

RTF, which was developed by Microsoft, is a file format used to display documents [16]. A standard RTF file consists only of 7-bit ASCII characters. RTF consists of control words, control symbols, and groups.

Figure 2 shows an example of RTF code. Braces (`{` and `}`) define a group, and groups can be nested. A backslash starts a RTF control code. A valid RTF document contains a group that starts with the `\rtf` control code. The first character of RTF code is `{`. The end of file marker (EOF) is `}`, corresponding to the first `{`. Document processors usually do not process data after the EOF.

```
{\rtf
Hello,\par
{\b world}!\par
}
```

Figure 2: Example of an RTF file.

2.1.2 Anomalous Structures

AS1:Attached data after EOF

An EOF should be located at the end of the file. However, the RTF files in the specimens often had an EOF in the middle of the file, after which an executable file was embedded.

2.2 CFB

2.2.1 Structure

This section describes the CFB specification.

CFB, which was developed by Microsoft, is a structured storage compound file implementation [14], also known as OLE (Object Linking and Embedding [15]) or COM (Component Object Model [12]). DOC (used by Microsoft Word), XLS (used by Microsoft Excel) and PPT (used by Microsoft PowerPoint) files have a CFB file format. OOXML [10] (DOCX, XLSX, and PPTX) files are usually zip containers that are encrypted following the CFB file format. JTD and JTDC files too follow a CFB file format. These are used in the Japanese Word Processor "Ichitaro," which is developed by JustSystems. "Ichitaro" is widely used in Japan, making it a common target when a vulnerability is found.

A CFB file is a single file that contains a nested hierarchy of storages and streams. Figure 3 shows the CFB file hierarchy, which is analogous to a file system hierarchy; storages are analogous to directories, and streams are analogous to files.

Figure 4 shows a CFB structure. A CFB file is divided into a 512-byte header and equal-length sectors. Subsequent sectors are identified by a 32-bit non-negative integer called the sector number. A stream is divided and stored into sectors.

A group of sectors can form a sector chain, a linked list of sectors forming a logical byte array. In a sector chain, a sector number is used to identify the next sector in the chain. "-1" and "-2" are special sector numbers. "-1" is used to represent free sectors, and "-2" is used to represent chain termination. The left side of Fig. 4 shows two sector chains. A sector chain (blue) starts at sector #2, and ends at sector #8. Another sector chain (red) starts at sector #5, and ends at sector #7.

Sector chains are defined in a FAT (File Allocation Table). DE (Directory Entry) manages the name, size, and

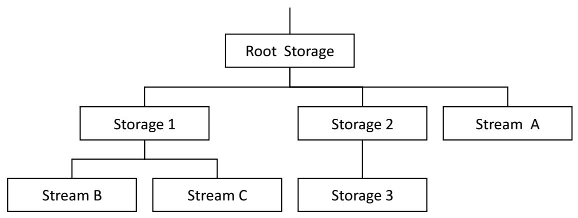


Figure 3: CFB hierarchy.

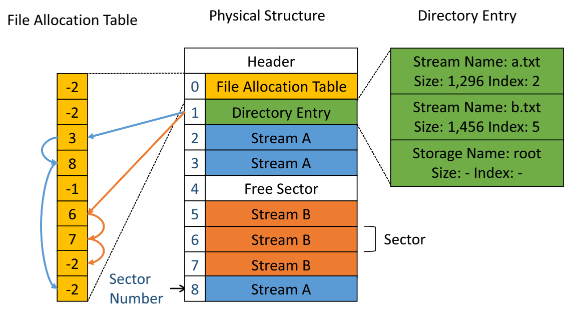


Figure 4: CFB structure.

parent-child relation of each stream or storage. The position of the FAT and the DE are defined in the header.

2.2.2 Anomalous Structures

AS2: Anomalous file size

The file size should equal 512 (the header size) plus a multiple of the sector size. We define $\mathrm{Size}_{\mathrm{file}}$ as the file size and $\mathrm{Size}_{\mathrm{sector}}$ as the sector size. The following equation should thus hold:

$$(\mathrm{Size}_{\mathrm{file}} - 512) \bmod \mathrm{Size}_{\mathrm{sector}} = 0. \tag{1}$$

In the CFB files in the specimens, the size of the embedded executable file rarely equals a multiple of the sector size, because the executable file is forcibly included in the CFB file. Therefore, Eq. (1) is usually not established.

AS3: Data not referenced by the FAT

As we mentioned, sector numbers are 32-bit (4-byte) non-negative integers. Therefore, one sector corresponding to the FAT can manage $\mathrm{Size}_{\mathrm{sector}} \div 4$ sectors. We define $\mathrm{Count}_{\mathrm{FAT}}$ as the number of sectors corresponding to the FAT and $\mathrm{Size}_{\mathrm{FAT}}$ as the maximum size of sectors managed by the FAT. The $\mathrm{Size}_{\mathrm{FAT}}$ is obtained by the following equation:

$$\mathrm{Size}_{\mathrm{FAT}} = \mathrm{Count}_{\mathrm{FAT}} \times \mathrm{Size}_{\mathrm{sector}} \div 4. \tag{2}$$

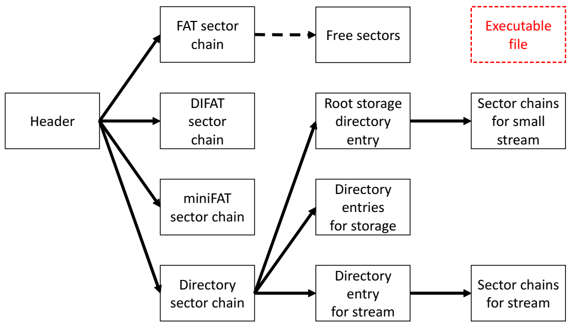


Figure 5: Example of a sector hierarchy.

All sectors are managed by the FAT. Therefore, the difference between the file size and 512 (the header size) should be less than or equal to Size_{FAT}. The following formula should thus hold:

$$\text{Size}_{\text{file}} - 512 \leq \text{Size}_{\text{FAT}}. \tag{3}$$

In CFB files in the specimens, the executable file is forcibly included into document files, ignoring FAT rules, so the file size exceeds Size_{FAT} and Eq. (3) does not hold.

AS4: Free sector in the last sector

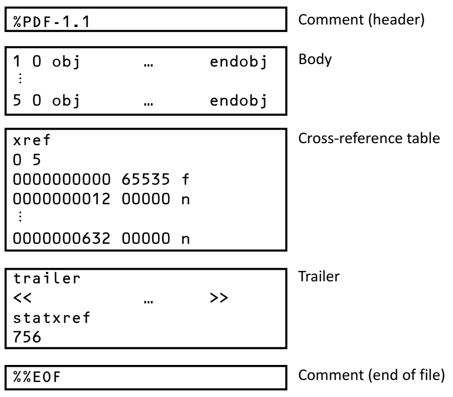
Free sectors are not generally processed by document processors, allowing use of free sectors to store hidden data. Specific locations for free sectors are not defined. We analyzed various CFB files not containing executable files, and found no CFB files whose last sector is a free sector, which should be in the middle of a CFB file. In the CFB files in the specimens, however, last sectors were free sectors.

AS5: Unaccounted-for sectors

A CFB file consists of six sector types, namely FAT, DIFAT (Double-Indirect FAT), miniFAT, DE, stream, and free sector. The DIFAT manages sectors corresponding to the FAT. MiniFAT sectors manage streams whose size is less than a constant size, defined in the header.

A CFB file can be regarded as a hierarchy of sectors (Fig. 5), with the header as the hierarchy root. Generally, all sectors are located within this sector hierarchy.

We can usually classify sectors according to their type, but in CFB files in the specimens, sectors containing the executable file appear outside the hierarchy.



```
%PDF-1.1

1 0 obj        …        endobj
⋮
5 0 obj        …        endobj

xref
0 5
0000000000 65535 f
0000000012 00000 n
⋮
0000000632 00000 n

trailer
<<             …        >>
statxref
756

%%EOF
```

Figure 6: Structure of a PDF file.

2.3 PDF

2.3.1 Structure

This section provides an overview of the PDF specification.

PDF, which is developed by Adobe Systems, is a file format for document display. The PDF specification was officially released as an open standard on July 1, 2008, and is now published by the International Organization for Standardization as ISO 32000-1:2008 [9]. Adobe Systems has extended the PDF specification as part of its “Adobe Extensions” [1].

A conforming PDF file should be constructed of four elements (Fig. 6):

- Comments, which have no semantics (except for `%PDF-n.m` and `%%EOF`)
- A body containing the objects that make up the document contained in the file
- A cross-reference table containing information about indirect objects in the file
- A trailer giving the location of the cross-reference table and of certain special objects within the file body

The trailer of a PDF file enables a conforming reader to quickly find the cross-reference table and certain special objects. The last line of the file should contain only the end-of-file marker, `%%EOF`. The two preceding lines should contain, once per line and in order, the keyword `startxref` and the byte offset from the beginning of the file to the beginning of the `xref` keyword in the last cross-reference section. The `startxref` line should be preceded by the trailer dictionary, consisting of the keyword `trailer`.

```
4 0 obj
<</Length 24 /Filter /ASCIIHexDecode>>
stream
48656C6C6F2C576F726C6421
endstream
endobj
```

Figure 7: Example of an object.

Table 4: Examples of standard filters.

ASCII85 Decode	Decodes data encoded in an ASCII base-85 representation, reproducing the original binary data.
ASCIIHex Decode	Decodes data encoded in an ASCII hexadecimal representation, reproducing the original binary data.
DCT Decode	Decompresses data encoded using a discrete cosine transform technique based on the JPEG standard, reproducing image sample data that approximates the original data.
Flate Decode	Decompresses data encoded using the zlib/deflate compression method, reproducing the original text or binary data.
JBIG2 Decode	Decompresses data encoded using the JBIG2 standard, reproducing the original monochrome image data.

Objects

A PDF file has nine basic object types: Boolean values, integer and real numbers, strings, names, arrays, dictionaries, streams, and null objects.

A stream object, like a string object, is a sequence of bytes. A stream should consist of a dictionary followed by zero or more bytes bracketed between the keywords `stream` (followed by a newline) and `endstream` (Fig. 7). One option when reading stream data is to decode it using a filter to reproduce the original non-encoded data. Whether to do so and which decoding filter or filters to use is specified in the stream dictionary. The standard filters are summarized in Table 4.

Objects may be labeled so that other objects can refer to them. A labeled object is called an indirect object. The definition of an indirect object in a PDF file consists of its object number and generation number (separated by white space), followed by the value of the object bracketed between the keywords `obj` and `endobj` (Fig. 7).

Document Structure

A PDF document can be regarded as a hierarchy of objects contained in the body section of the PDF file. Figure 8 illustrates the structure of an object hierarchy. At the root of the hierarchy is the document's catalog dictionary. Generally, all objects are located within the hierarchy.

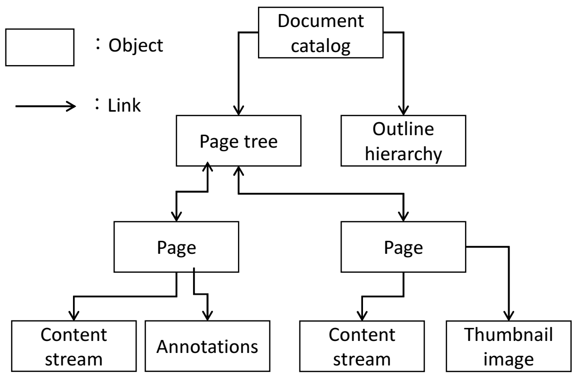


Figure 8: Structure of a PDF document.

Encryption

PDF documents support encryption to protect their contents from unauthorized access. Encryption-related information is stored in a document's encryption dictionary, which should be the value of the `Encrypt` entry in the document's trailer dictionary. Encryption applies to most strings and streams in the document. Encryption is not applied to object types such as integers and Boolean values, which are used primarily to convey information about the document's structure rather than its content. Leaving these values unencrypted allows access to the objects within a document.

However, an encrypted file containing object streams denies access to objects within the object streams. An object stream is a stream in which a sequence of indirect objects may be stored. The purpose of object streams is to allow indirect objects other than streams to be stored more compactly by using the facilities provided by stream compression filters. In an encrypted file (i.e., all object streams are encrypted), strings occurring anywhere in an object stream should not be separately encrypted. Therefore, an encrypted object stream denies access to objects within it.

2.3.2 Anomalous Structures

AS6: Unaccounted-for sections

A basic conforming PDF file is constructed of four element types (comment, body, cross-reference table, and trailer). However, PDF files in the specimens have unaccounted-for sections that are not of any of these four types.

AS7: Unreferenced objects

PDF documents can be regarded as a hierarchy of objects contained in the body section of the PDF file. In PDF files in the specimens, embedded executable files were camouflaged as an object and forcibly included into the PDF file, ignoring the hierarchy rules so that they appear outside the hierarchy.

AS8: Camouflaged stream

Generally, we can decode encoded streams without problems. Additionally, the length of the data processed for decoding should be equal to the length of the stream data.

In PDF files in the specimens, a stream contains an executable file, and stream decoding was nonstandard. The details are as follows.

AS8-1: Camouflaged filter

Filters corresponding to camouflaged streams are sometimes incorrect, causing stream decoding to fail. We suspect the following reason for why the filter was camouflaged by attackers.

Executable files are similar to compressed streams, in that data entropy is high in both cases. Entropy is commonly associated with the amount of order or disorder. As a measure of disorder, the higher the entropy is, the greater the disorder is. A camouflaged stream is thus hard to distinguish from a compressed stream in terms of entropy. Therefore, a camouflaged filter such as FlateDecode is widely used in Specimens 1.

Streams with a camouflaged filter are usually not processed by document processors. If the software processes the stream, the displayed content gets garbled or the software runs incorrectly. Therefore, streams with camouflaged filter often appear outside the PDF hierarchy (see AS7).

AS8-2:Extra data after EOD

Even if a camouflaged stream contains extra data such as an executable file, the stream can be decoded without problems, in some cases because of an end-of-data (EOD) marker, as follows.

Most filters are defined such that data are self-limited; they use an encoding scheme that limits the length of data with an explicit EOD marker. Document processors usually do not process data after the EOD, so most streams with added extra data can still be decoded without problems. Taking advantage of this feature, however, attackers can add an executable file to the stream.

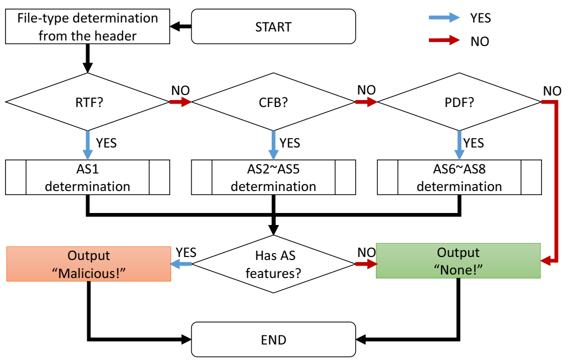


Figure 9: Flow of checking a document file with o-checker.

3 O-checker

We created a tool, "o-checker," to detect the eight anomalous structures described above. O-checker is a command line program written in Python that needs only one argument, a target file path.

Figure 9 shows the flow of checking a document file with o-checker. O-checker reads a file header, determines the file type, and examines each anomalous structure. The details of the detection are as follows.

3.1 RTF

AS1 determination method

O-checker reads one byte at a time and checks whether the character signifies an EOF. When there is data after the EOF, o-checker determines that the document file has an AS1 feature.

3.2 CFB

AS2 determination method

A sector shift (2 bytes) field is present in the header at file offset 30. A sector shift specifies the sector size as a power of 2. This field must be set to 0x0009 ($\text{Size}_{\text{sector}} =$ 512 bytes) or 0x000C ($\text{Size}_{\text{sector}} = 4096$ bytes). When Eq. (1) is not satisfied, o-checker determines that the document has an AS2 feature.

AS3 determination method

The number of FAT sectors (4 bytes) field is present in the header at file offset 44. This integer field contains the number of FAT sectors in the CFB file. We define $\text{Count}_{\text{FAT}}$ as this value, and calculate Size_{FAT} using Eq. (2). When inequality (3) is not satisfied, o-checker determines that the document has an AS3 feature.

AS4 determination method

We define the sector number of the last sector of a CFB file as n. The CFB file should have a 512-byte header and $n+1$ sectors, so the following formula should hold:

$$\text{Size}_{\text{file}} = 512 + (n+1) \times \text{Size}_{\text{sector}}. \tag{4}$$

We solve this equation for n. When the FAT value of n-th sector is -1 (indicating a free sector), o-checker determines that the document file has an AS4 feature.

AS5 determination method

We can detect unaccounted-for sectors by constructing the hierarchy of sectors, starting from the header. We then check all sectors to determine whether these sectors appear in the hierarchy. To maximize o-checker's speed, we focus on differences between the number of classified sectors and that of sectors of the file. When we find a difference, o-checker determines that the document file has an AS5 feature.

We define $\text{Count}_{\text{real}}$ as the number of sectors in the CFB file. The file size should be 512 bytes (the header size) plus the sector size multiplied by $\text{Count}_{\text{real}}$. Thus, the following formula should hold:

$$\text{Size}_{\text{file}} = 512 + \text{Count}_{\text{real}} \times \text{Size}_{\text{sector}}. \tag{5}$$

We solve this equation for $\text{Count}_{\text{real}}$.

We define the following variables:

- $\text{Count}_{\text{FAT}}$ is the number of sectors comprising the FAT.
- $\text{Count}_{\text{miniFAT}}$ is the number of sectors comprising the miniFAT.
- $\text{Count}_{\text{DIFAT}}$ is the number of sectors comprising the DIFAT.
- Count_{DE} is the number of sectors comprising the DE.
- $\text{Count}_{\text{Streams}}$ is the number of sectors comprising all streams.
- $\text{Count}_{\text{free}}$ is the number of free sectors.
- $\text{Count}_{\text{classified}}$ is the number of classified sectors.

The following formula should hold:

$$\begin{aligned}\text{Count}_{\text{classified}} = {} & \text{Count}_{\text{FAT}} + \text{Count}_{\text{miniFAT}} + \text{Count}_{\text{DIFAT}} \\ & + \text{Count}_{\text{DE}} + \text{Count}_{\text{Streams}} + \text{Count}_{\text{free}}.\end{aligned}$$

We can obtain each variable as follows.

- $\text{Count}_{\text{FAT}}$ (4 bytes) is present in the header at file offset 44.
- $\text{Count}_{\text{miniFAT}}$ (4 bytes) is present in the header at file offset 64.
- $\text{Count}_{\text{DIFAT}}$ (4 bytes) is present in the header at file offset 72.

The header has only the sector number (4 bytes) of the first sector of the DE. Count_{DE} is obtained in three steps: The first step is reading the 4-byte integer from the header at file offset 48. The second step is getting the whole picture of the DE from the FAT. The final step is counting sectors comprising the DE.

Obtaining $\text{Count}_{\text{Streams}}$ is more complex. The first step is getting the whole picture of the DE, as described above. The second step is getting all the stream entries from the DE. Next, we obtain the number of sectors for each stream. The final step is calculating the total number of sectors for each stream.

We obtain the number of stream sectors as follows: We define Size_n as the size of n-th stream and define Count_n as the number of sectors for the n-th stream. The value (4 bytes) of Size_n is present in the n-th entry at offset 120. When Size_n is less than a constant value, the stream is stored as a "root entry" stream, and Count_n is "0." We define the constant value as $\text{Size}_{\text{mini}}$. $\text{Size}_{\text{mini}}$ is present in the header at file offset 56. When Size_n is more than $\text{Size}_{\text{mini}}$, Count_n equals Size_n divided by $\text{Size}_{\text{sector}}$ and rounded up to the nearest integer. Therefore, Count_n is expressed as

$$\text{Count}_n = \begin{cases} 0 & (\text{Size}_n < \text{Size}_{\text{mini}}) \\ \lceil \text{Size}_n \div \text{Size}_{\text{sector}} \rceil & (\text{Size}_n \geq \text{Size}_{\text{mini}}) \end{cases}. \tag{6}$$

We obtain $\text{Count}_{\text{free}}$ by counting the number of sectors whose FAT value is -1.

$\text{Count}_{\text{real}}$ should equal $\text{Count}_{\text{classified}}$. When a difference between $\text{Count}_{\text{real}}$ and $\text{Count}_{\text{classified}}$ appears, o-checker determines that the document file has an AS5 feature.

3.3 PDF

AS6 determination does not depend on the condition of a given PDF file. When a PDF file is encrypted, AS7 and AS8 determination are sometimes not applicable to the file.

Figure 10 shows the flow of checking a PDF file with o-checker. First, it checks whether the file has an AS6 feature. Next, o-checker checks whether the file is encrypted according to the trailer dictionary. If the file is not encrypted, o-checker checks whether the file has an AS7 or AS8 feature.

Generally, encrypted PDF files can be decrypted by two types of passwords, a user password or an owner password. When the user password is an empty string,

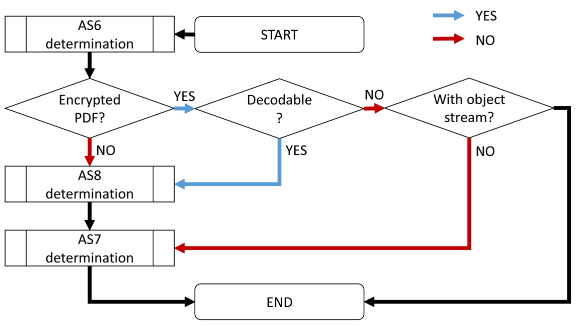


Figure 10: Flow of checking a PDF file with o-checker.

it suppresses prompting for a password when the file is opened. We usually do not mind whether the file is encrypted, because many PDF files are encrypted to avoid detection based on pattern matching, but using an empty string as the user password.

If the input file is encrypted, o-checker tries to decrypt the file using an empty string as the user password. O-checker can currently handle four types of encryption method, namely 40-bit RC4, 128-bit RC4, 128-bit AES, and 256-bit AES. When o-checker succeeds in decryption, it checks whether the file has an AS7 or AS8 feature.

When o-checker fails the decryption, it searches for object streams. If no object stream is present in the file, o-checker checks whether the file has an AS7 feature. If an object stream is present in the file, evaluation of the PDF file is finished.

The details of each evaluation method in Fig. 10 are described below.

AS6 determination method

O-checker reads files one byte at a time, classifying them according to the following four types:

- A comment is a line starting with a % character.
- A body consists of indirect objects, bracketed between the keywords `obj` and `endobj`.
- A cross-reference table is the lines starting from `xref`.
- A trailer is the lines starting from `trailer`.

When unclassified data is present in the file, o-checker determines that the document file has an AS6 feature.

AS7 determination method

O-checker constructs a hierarchy of indirect objects for the body section of the PDF file to search for unreferenced objects. First, o-checker reads all indirect objects in the PDF file. Next, o-checker finds all links to indirect objects to search for unreferenced objects. If o-checker finds an unreferenced object whose size exceeds 512 bytes, it determines that the document file has an AS7 feature.

We exclude small (< 512 bytes) objects from the judgment process, because these cannot contain an executable file. In Portable Executable (PE) files, the header consists of an MS-DOS 2.0 Section, unused (null) data, a PE header, and so on [13]. The size of the header is at least 512 bytes, so we set 512 as the criterion size.

AS8 determination method

O-checker can reveal camouflaged filters that use FlateDecode, ASCIIHexDecode, ASCII85Decode, DCTDecode, or JBIG2Decode. It first tries to decode streams using FlateDecode, ASCIIHexDecode, and ASCII85Decode. If the decoding process fails, o-checker determines that the document file has an AS8 feature.

In the case that a stream uses FlateDecode, DCTDecode, or JBIG2Decode, o-checker searches for extra data in the stream. If there is a difference between the position of the EOD marker and the position of the last byte of the stream, o-checker determines that the document file has an AS8 feature.

4 Evaluation

We evaluated o-checker's effectiveness in three experiments. The following summarizes our experiments:

- Experiment 1: True positive rate to malicious files containing an executable file like those used in targeted email attacks
- Experiment 2: False negative rate for benign files
- Experiment 3: True positive rate for all malicious files

Table 5 shows our experimental environment. We run o-checker on a virtual machine. The top of Table 5 shows specifications of the host machine platform, which consists of a computer with a Core i5-3450 3.1 GHz CPU and Windows 7 SP1. We run the virtual machine in VMware Workstation 9. The bottom of Table 5 shows specifications of the virtual machine platform, a dual-core CPU machine running Windows XP SP3. We used Python 2.7.6.

The details of the experiments are as follows.

Table 5: Experimental environment.

CPU	Core i5-3450 3.1 GHz
Memory	8.0 GB
OS	Windows 7 SP1
Virtualization software	VMware Workstation 9
Memory (VM)	512 MB
OS (VM)	Windows XP SP3
Interpreter (VM)	Python 2.7.6

Table 6: Summary of Specimens 2.

Type	Ext.	Num.	Avg. Size (KB)	Camouflaged
RTF	rtf	44	554.1	40
CFB	doc	35	193.8	2
	xls	15	277.3	0
	pps	1	1210.0	0
	jtd	25	437.0	0
PDF	pdf	7	753.0	0
Total	-	127	415.2	42

4.1 Experiment 1

Experiment 1 evaluates o-checker's true positive rate (TPR) for document files containing an executable file.

We classified the eight anomalous structures from Specimens 1. We obtained targeted emails that various Japanese organizations received in 2013 and 2014. We mechanically extracted the document files from these emails, classifying them according to the file name. And removed duplicate files as identified by hash key. We confirmed that each specimen contains an executable file. This resulted in 127 specimens, which we call "Specimens 2."

Table 6 summarizes Specimens 2. The extensions of the specimens are sometimes camouflaged, so we classified specimens according to their header. The "pps" file type is associated with Microsoft PowerPoint.

Table 7 shows the vulnerabilities exploited by Specimens 2. The most exploited vulnerability is MS12-027, and 23 specimens exploited a zero-day vulnerability, meaning a software vulnerability that is unknown to the vendor. The total for the Rates column (right side of Table 7) is more than 100% because some specimens exploited multiple vulnerabilities.

We input the specimens to o-checker, then calculated TPR and measured the running time. Additionally, we compared the ability of o-checker with three popular brands of anti-virus software. Most anti-virus software can detect well-known malware by updating a pattern file. We wished to evaluate the TPR of anti-virus software against unknown malware, so we updated anti-virus software every day and performed evaluations immediately after obtaining the specimens.

Table 7: Vulnerability used by Specimens 2.

Vulnerability	Num.	Rate
MS10-087	2 / 127	1.6%
MS12-027	71 / 127	55.9%
MS14-017	3 / 127	2.4%
JS13003	15 / 127	11.8%
JS14003	19 / 127	15.0%
APSB10-21	4 / 127	3.2%
APSB11-08	4 / 127	3.2%
APSB11-30	4 / 127	3.2%
APSB12-22	1 / 127	0.8%
APSB13-07	3 / 127	2.4%
None	3 / 127	2.4%
Unknown	6 / 127	4.7%

Table 8: TPR of o-checker against Specimens 2.

Type	Ext.	Detection	TPR	Avg. Time
RTF	rtf	41 / 44	93.2%	0.226 s
CFB	doc	34 / 35	97.1%	0.170 s
	xls	15 / 15	100.0%	0.184 s
	pps	0 / 1	0%	0.171 s
	jtd	25 / 25	100.0%	0.194 s
PDF	pdf	7 / 7	100.0%	1.382 s
Total	-	122 / 127	96.1%	0.263 s

Result

Table 8 shows the TPR of o-checker against Specimens 2. The "Detection" column in this table shows the number of detected files divided by the number of specimens. O-checker could detect 96.1% of all specimens.

Table 9 shows the results of comparison with popular anti-virus software. The "Detection" column in this table is the same as that in Table 8. None of the popular anti-virus software could detect even half of the specimens. Detection characteristics varied for each anti-virus software. A combination of all three could detect only 50.4% of the specimens.

Table 9: Comparing TPR with anti-virus software.

	detection	TPR
o-checker	122 / 127	96.1%
T's AV	46 / 127	36.2%
S's AV	30 / 127	23.6%
M's AV	23 / 127	18.1%
T&S&M's AV	64 / 127	50.4%

Table 10: Summary of Specimens 3.

Type	Ext.	Num.	Avg. Size (KB)
RTF	rtf	199	516.2
CFB	doc	1,195	106.1
	xls	298	191.7
PDF	pdf	9,109	101.7
Total	-	10,801	112.3

Table 11: FPR of o-checker against Specimens 3.

Type	Ext.	Detection	FPR
RTF	rtf	0 / 199	0.0%
CFB	doc	2 / 1,195	0.2%
	xls	14 / 298	4.7%
PDF	pdf	19 / 9,109	0.2%
Total	-	35 / 10,801	0.3%

4.2 Experiment 2

Experiment 2 evaluates o-checker's false positive rate (FPR) against benign document files.

We prepared 10,801 specimens from a web site, "Contagio" [17]. Contagio distributes benign files for signature testing and research from various open sources. Some of the specimens for some reason had added HTML data in their headers, so we removed files with a mismatch between the file extension and the header. These processed specimens are denoted as Specimens 3. Table 10 summarizes Specimens 3.

We scanned these specimens using o-checker. When o-checker detected anomalous structures, we defined the detection as a false positive.

Results

Table 11 shows the FPR of o-checker against Specimens 3. The FPR against all the specimens is 0.3%. However, note that the FPR against xls files in the specimens is a relatively high 4.7%.

4.3 Experiment 3

Experiment 3 evaluates o-checker's detection rate against malicious document files. We prepared 227 specimens for Experiment 3, denoted as Specimens 4. Table 12 summarizes Specimens 4. Some of Specimens 4 do not contain executable files, unlike the files used in Experiment 1.

We obtained Specimens 4 from VirusTotal [22]. The search conditions specified files that exploit a CVE vulnerability, were obtained in 2013 or 2014, and have a file extension of rtf, doc, xls, ppt, or pdf. We mechanically obtained specimens from the search results. We classified specimens according to the file content in the same manner as Experiment 1.

We scanned the specimens using o-checker, then calculated the resulting TPR. We did not measure the TPR of anti-virus software for the following reason. VirusTotal recorded these specimens, and after several months we obtained them. Because we updated anti-virus software every day, anti-virus software detected almost all anomalous structures in Specimens 4. Thus, we could not obtain the TPR of anti-virus software against unknown malware.

We compared the ability of o-checker with OfficeMalScanner [2] (OMS) v0.58 in place of anti-virus software. OMS mainly scans files for generic shell code patterns. OMS can scan both CFB and RTF files. For CFB files, we used `OfficeMalScanner.exe` with the `SCAN` and `BRUTE` options. For RTF files, we used the `RTFScan.exe` tool included in OMS with the `SCAN` option, which scans a target file for generic shell code patterns, an embedded CFB signature, or an embedded executable file. The `BRUTE` feature scans for an encrypted CFB signature or an encrypted executable file.

We also examined the ability of o-checker combined with OMS. Figure 11 shows the flow of combined detection. In short, when o-checker or OMS detects a file, this method determines that the file is malicious. Because OMS does not support PDF files, we excluded PDF files from Specimens 4. Specimens 5 denotes these excluded specimens.

Table 12: Summary of Specimens 4.

Type	Ext.	Num.	Avg. Size (KB)
RTF	rtf	69	487.7
CFB	doc	61	259.0
	xls	9	298.4
	ppt	2	480.5
PDF	pdf	86	653.5
Total	-	227	481.5

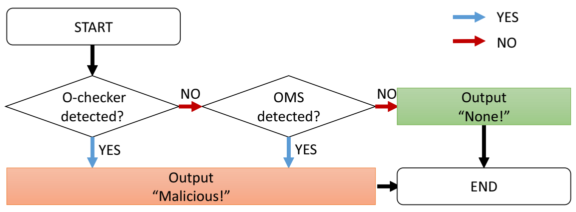


Figure 11: Flow of o-checker combined with OMS.

Table 13: TPR of o-checker and OMS against Specimens 4.

Type	Ext.	o-checker	OMS
RTF	rtf	34 / 69	12 / 69
CFB	doc	44 / 61	31 / 61
	xls	2 / 9	3 / 9
	ppt	0 / 2	0 / 2
PDF	pdf	40 / 86	-
Total	-	120 / 227	46 / 141
TPR		52.9%	32.6%

Table 14: TPR of o-checker combined with OMS against Specimens 5.

Type	Ext.	o-checker	OMS	Combination
RTF	rtf	34 / 69	12 / 69	39 / 69
CFB	doc	44 / 61	31 / 61	51 / 61
	xls	2 / 9	3 / 9	5 / 9
	ppt	0 / 2	0 / 2	0 / 2
Total	-	80 / 141	46 / 141	95 / 141
TPR		56.7%	32.6%	67.3%

Results

Table 13 shows the TPR of o-checker and OMS, with the "OMS" column showing detection by `OfficeMalScanner.exe` or `RTFScan.exe`. The TPR of o-checker is 52.9%, and that of OMS is 32.6%.

Table 14 shows the TPR of o-checker combined with OMS against Specimens 5. The "Combination" column in this table shows detection by o-checker or OMS. The TPR of o-checker alone is 56.7%, and the combination increases the TPR to 67.3%.

5 Discussion

5.1 Analysis of detection leakage for Specimens 2

In Experiment 1, o-checker could not detect five specimens in Specimens 2. We analyzed these specimens with a focus on how the executable file was embedded. The results revealed three methods by which detection by o-checker was avoided. The details of these methods follow.

5.1.1 Embedding by document processors

We can embed any executable file into a document file using document processors by transforming the executable file into an OLE object and including it into the document file according to the specification. The resulting document file has no anomalous structures, so o-checker cannot detect any anomalies.

However, execution of the embedded file requires manual operations such as a double-clicking the OLE object and verifying a warning dialog. These operations reduce the success rate of targeted attacks, so it is rare that executable files are embedded into document files by this method.

5.1.2 Embedding into shell code

Generally, exploit code needs to be processed by document processors to exploit vulnerabilities in them. In other words, most of the exploit code is present in document files according to the specification. In addition, most exploit codes contain the entirety of the shell code. When an executable file is embedded into a large shell code, the document file does not have the anomalous structures we described, so o-checker rarely detects this kind of file.

However, the possible size of shell code is often limited, in turn limiting embedded executable files to even smaller sizes. Thus, executable files are rarely embedded in shell codes.

5.1.3 Camouflaged Stream in a CFB file

When a stream in a CFB file contains an executable file, the CFB file has none of the anomalous structures described above. Thus, o-checker cannot detect anomalous structures in the CFB file.

If document processors process the stream, the displayed content is garbled, or the software leads to strange behavior. In this case, we can predict that the stream appears outside the hierarchy of streams, like an AS7 feature in a PDF file. We expect that we can detect anomalous structures in a CFB file by searching for unreferenced streams, like the AS7 determination method.

Additionally, there are many parse tools for PDF files, but relatively few for CFB files. Attackers need to learn the specifications of the CFB format to embed a stream containing an executable file into a CFB file. In the case of CFB files, it is rare that executable files are camouflaged as streams.

5.2 False positives in Experiment 2

O-checker generated false positives for 35 benign files in Experiment 2. We analyzed these files with a focus on deviations from file format specifications. As the result, we can classify these false positives into three types: data corruption, injection of extra data, and benign unreferenced objects. The details follow.

5.2.1 Data corruption

One cause of misdetection is file fragmentation. Fragmented document files violate the specification, creating some of the anomalous structures we described.

Document files made by document processors usually are not fragmented, because document processors cannot display the contents of the fragmented file. Although the fragmented file might be a benign file, detecting it as anomaly file may be a good thing from the aspect of data integrity.

5.2.2 Injection of extra data

Another cause of misdetection is the injection of extra data at the end of the file. In all cases, the size of extra data was less than 4 KB. The size of executable files in Specimens 3 was more than ten times this, so we can likely avoid the misdetection through filtering. The drawback to this is that attackers might then use an object that passes the filter.

Then again, files made by document processors usually do not contain extra data. Although a document file containing extra data might be benign, detecting it as anomaly file might be a good thing from the aspect of data integrity.

5.2.3 Containing benign unreferenced object

Benign unreferenced objects can also result in misdetection. Twelve PDF files in Specimens 3 have unreferenced objects (AS7), for unknown reasons. These unreferenced objects are similar to the configuration information of a page of the document. The size of each object was less than 4 KB, as in the case of extra data described above, so similar pros and cons of filtering these anomalies hold here too.

5.3 Long-term detection rate of o-checker

When we analyzed the document files used in targeted email attacks from 2009 to 2012, we found eight anomalous structures, and we created o-checker to detect these anomalous structure. When we checked malicious document files used in targeted email attacks in 2013 and 2014 with o-checker, we could detect these files with a fairly high rate.

Generally, the structures of document files are complex, so decoders that export data from a valid document file are also complex. In turn, a document file containing an executable file according to the document specification will often have complex shell code. When the size of shell code is limited, the shell code cannot contain an executable file, and furthermore the decoder contained in the shell code must have simple code. Thus, executable files are almost always forced into document files, ignoring the specification. These document files have the anomalous structures we mentioned, and we can detect these files.

There are far fewer stealth techniques for hiding executable files than vulnerabilities of document processors. Additionally, the pace of transition of document file format specifications is much slower than that of document processor updates. Accordingly, we assert that o-checker can continue detecting malware with a fairly high detection rate for fairly long period.

5.4 Application of o-checker

O-checker could detect 96.1% of Specimens 2 without virus pattern files in Experiment 1. Furthermore, the check speed of o-checker was 0.263 sec per file. File checks using malware sandbox analysis generally take a few minutes, so o-checker is comparatively fast. If we combine o-checker with a mail server, we can detect document files containing executable files passing through the mail server.

However, malware is not only delivered as document files containing executable files. Thus, we used o-checker to examine malicious document files recorded in VirusTotal in Experiment 3, by which we could detect 52.9% of the malicious files. Some of the specimens have none of the anomalous structures we mentioned. Therefore, improving detection rates will require combining other methods with o-checker.

In the case of Experiment 3, the detection rate of OMS was lower than that of o-checker. However, OMS sometimes detected specimens that could not be detected by o-checker. Thus, the detection rate of o-checker combined with OMS (67.3%) is higher than the detection rate of o-checker alone (56.7%).

5.5 Limitations of o-checker

O-checker is file format dependent, and can only examine RTF, CFB, or PDF files. In particular, o-checker cannot examine Office Open XML (OOXML) [10] files. However, we have not seen OOXML files containing executable files and exploit code until very recently, for two reasons.

First, the OOXML specification has a mechanism for detecting extra files. An OOXML file is a zip container containing many files. The specification of OOXML defines a "relationship" that describe usages of all the files. A file without a relationship is considered to be an extra file, and document processors cannot open an OOXML file containing extra files such as an executable file.

The second reason is that zip decoders require a large amount of code. The size of shell code is often limited to a small size, so zip decoders embedded within shell code are quite rare.

For these reasons, we have obtained OOXML files containing executable files only very recently, the first appearing in 2015. This example exploited a vulnerability in Microsoft Office via an EPS (encapsulated Postscript) file containing an executable file. Should such approaches gain popularity, we should consider strategies for scanning OOXML files.

6 Related Work

We conducted a static analysis, examined only document files without running embedded executable files. Indeed, we did not consider executable files at all, instead focusing only on document specifications. References [19] and [18] are previous studies by the authors relevant to this paper. Below is an overview of other related works.

6.1 Methods focusing on program codes

Laskov used machine learning to detect malicious PDF files in a fast scan [11]. The method focuses on JavaScript in PDF files. Because the method uses a supervised learning model, it needs samples for learning and the detection rate depends on these samples. In contrast, our method does not require learning samples.

Laskov's method can detect whether a PDF file contains an executable file. However, exploit codes use not only JavaScript but also Flash, font, and image files. When searching for exploit code, it is important to consider various types of exploit code. Our method cannot detect exploits in a document file not containing an executable file, but our method does not depend on the type of exploit codes.

OfficeMalScanner [2] (OMS) is an analysis tool for document files. OMS scans entire files for generic shell code patterns, an embedded CFB signature, or an embedded executable file. However, OMS rarely detects encrypted shell code or embedded files. This is why the detection rate of OMS is lower than that of o-checker in Experiment 3. Our method is not affected by encryption, because it focuses on deviation from file format specifications.

6.2 Methods focusing on document file structures

Hyukdon suggested a tool for analyzing extra data in a CFB file [8]. The tool examines four areas of the CFB file to search for hidden data. The four areas can contain free sectors, so o-checker also examines free sectors. A document file containing extra data might be malicious, but benign CFB files often contains free sectors. Thus, if we use Hyukdon's tool to detect malicious CFB files, the tool would return many false positives.

Our method distinguishes between benign free sectors and those containing an executable file, resulting in the low FPR of o-checker.

Xu et al. [23] and Srndic et al. [20] applied machine learning to detecting malicious PDF files, the former focusing on filters and the latter on document hierarchy. These methods can detect malicious PDF files not containing an executable file. Because these methods use a supervised learning model, they require samples for learning and the detection rate of these methods depends on these samples.

Our methods rarely detect malicious PDF files not containing an executable file. However, the document files used in targeted email attacks in Japan very often contain an executable file, and our method does not require samples for learning.

7 Conclusion

We proposed a tool for detecting document files containing an executable file. The tool, o-checker, examines document files according to the specification of the document format. We demonstrated the effectiveness of o-checker, with the result that it detected 96.1% of malicious specimens at an average rate of 0.263 seconds per file. There are far fewer stealth techniques for hiding executable files than vulnerabilities of document processors. Additionally, the pace of changes in document file format specifications is quite slower than that of document processor updates. Accordingly, we feel that o-checker can continue detecting malware with a fairly high detection rate for relatively long times.

Evaluation of the universal effectiveness of o-checker is a future task. O-checker could detect document files used in targeted attacks in Japanese organizations with fairly high detection rates. However, it is not clear that o-checker would be effective in other countries. Therefore, in future studies we will examine malicious document files obtained from organizations in other countries.

References

[1] ADOBE. PDF Reference and Adobe Extensions to the PDF Specification. `http://www.adobe.com/jp/devnet/pdf/pdf_reference.html`.

[2] BOLDEWIN, F. Analyzing MSOffice malware with OfficeMalScanner. `http://www.reconstructer.org/papers/Analyzing\%20MSOffice\%20malware\%20with\%20OfficeMalScanner.zip`.

[3] CVE DETAILS. Adobe Acrobat Reader : CVE security vulnerabilities, versions and detailed reports. `http://www.cvedetails.com/product/497/Adobe-Acrobat-Reader.html?vendor_id=53`.

[4] CVE DETAILS. Adobe Flash Player : CVE security vulnerabilities, versions and detailed reports. https://www.cvedetails.com/product/6761/Adobe-Flash-Player.html?vendor_id=53.

[5] CVE DETAILS. Microsoft Excel : CVE security vulnerabilities, versions and detailed reports. https://www.cvedetails.com/product/410/Microsoft-Excel.html?vendor_id=26.

[6] CVE DETAILS. Microsoft Powerpoint : CVE security vulnerabilities, versions and detailed reports. https://www.cvedetails.com/product/623/Microsoft-Powerpoint.html?vendor_id=26.

[7] CVE DETAILS. Microsoft Word : CVE security vulnerabilities, versions and detailed reports. https://www.cvedetails.com/product/529/Microsoft-Word.html?vendor_id=26.

[8] HYUKDON, K., YEOG, K., SANGJIN, L., AND JONGIN, L. A tool for the detection of hidden data in microsoft compound document file format. 08 Proceedings of the 2008 International Conference on Information Science and Security. 2008, pp. 141–146.

[9] ISO. ISO 32000-1:2008 Document management - Portable document format - Part 1：PDF1.7. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=51502.

[10] ISO. ISO/IEC 29500:2012:Information technology – Document description and processing languages – Office Open XML File Formats, 2012.

[11] LASKOV, P., AND SRNDIC, N. Static detection of malicious javascript-bearing pdf documents. In *ACSAC* (2011), ACM, pp. 373–382.

[12] MICROSOFT. Component Object Model (COM). https://msdn.microsoft.com/en-us/library/ms680573.aspx.

[13] MICROSOFT. Microsoft PE and COFF Specification. https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx.

[14] MICROSOFT. [MS-CFB]: Compound File Binary File Format. https://msdn.microsoft.com/ja-jp/library/dd942138.aspx.

[15] MICROSOFT. [MS-OLEDS]: Object Linking and Embedding (OLE) Data Structures. https://msdn.microsoft.com/en-us/library/dd942265.aspx.

[16] MICROSOFT. Rich Text Format (RTF) Specification, version 1.9.1. http://www.microsoft.com/en-us/download/details.aspx?id=10725.

[17] MILA, P. 16,800 clean and 11,960 malicious files for signature testing and research. http://contagiodump.blogspot.jp/2013/03/16800-clean-and-11960-malicious-files.html.

[18] OTSUBO, Y., MIMURA, M., AND TANAKA, H. Applying file structure inspection to detecting malicious pdf files. *Information Processing Society of Japan (IPSJ) Journal 55*, 10 (oct 2014), 2281–2289.

[19] OTSUBO, Y., MIMURA, M., AND TANAKA, H. Methods to detect malicious ms document file using file structure inspection. *Information Processing Society of Japan (IPSJ) Journal 55*, 5 (may 2014), 1530–1540.

[20] SRNDIC, N., AND LASKOV, P. Detection of malicious pdf files based on hierarchical document structure. In *NDSS* (2013), The Internet Society.

[21] TREND MICRO. Targeted Attack Campaigns and Trends: 2014 Annual Report. http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-targeted-attack-trends-annual-2014-report.pdf.

```
> python o-checker.py malware.doc
Malicious!
```

Figure 12: Example of o-checker input and output.

[22] VIRUSTOTAL. Virustotal. https://www.virustotal.com/.

[23] XU, W., WANG, X., ZHANG, Y., AND XIE, H. A fast and precise malicious pdf filter. Proceedings of the 22nd Virus Bulletin International Conference. 2012, pp. 14–19.

Appendix

A The usage manual of o-checker

This section briefly describes how to use o-checker.

A.1 How to install o-checker

O-checker is available from Black Hat USA 2016 website.

The requirements for using o-checker are as follows:

- Python 2.7.3 or later
- An OS that can run Python
- The PyCrypto package for Python (for encrypted PDF files)

A.2 The basic usage of o-checker

In normal use, the path to a target file is the only parameter passed to `o-checker.py`. Figure 12 shows an example of o-checker input and output. In this case, we ran `o-checker.py` against a malicious Microsoft doc file, which results in the `Malicious!` output. When the document file does not have any of the targeted anomalous structures, the output is `None!`.

A.3 Advanced usage of o-checker

O-checker is not only a detection tool but also an analysis tool that can describe the detailed structure of CFB or PDF document files. The details of usage are as follows.

A.3.1 Analyzing CFB files

O-checker contains `msanalysis.py` for analyzing CFB files. `msanalysis.py` scans a CFB file, and it outputs analysis logs describing the final determination of `Malicious!` or `None!`.

Figure 13 shows an example of output of `msanalysis.py` against a malicious doc file with the judgement option (`-j`). The result of the check is shown in Fig. 13 as `Malicious!`. The following describes the tool output:

```
> python msanalysis.py -j malware.doc
Compound File
1536
This is DocFile
Size of a sector: 512
Size of a short-sector: 64
Total number of sectors: 1
SecID of first sector of the dictionary stream 17
Minimum size of standard stream 4096
SecID of first sector of ssat 19
Total number of short-sectors: 1
0 Root Entry 20 stream size: 8064 composed size: 8192
1 U:Data 8 stream size: 4096 composed size: 4096
2 U:WordDocument 0 stream size: 4096 composed size: 4096
:
18 Empty -2 stream size: 0 composed size: 0
19 U:CompObj 124 stream size: 121 composed size: 128
suspicious file size!
00008800-000089FF:unused
00008A00-00008BFF:unused
:
0000FE00-0000FFFF:unused
00010000-000101FF:unused
suspicious unused sector!
file size: 140218
file size error!
header size: 1536
total composed size: 28672
Dictionary Stream size: 2560
unused sector 31232
unknown data: 107450
Null block size: 15360

Suspicious 2
Malicious!
run time: 0.0584909915924 sec
```

Figure 13: Example of msanalysis.py input and output.

- This doc file contains 20 directory entries (Nos. 0–19).
- There is data (at file offset 0x8800 to 0x101FF) not referred to in the FAT. (AS3)
- `Suspicious unused sector` means the last sector is a free sector. (AS4)
- The file size is 140,218 bytes. 140218 mod 512 = 442 (nonzero). `File size error!` means the file size is anomalous. (AS2)
- 107,450 bytes of data is unaccounted for. (AS5)

From the above, this doc file has AS2, AS3, AS4, and AS5 features, and an executable file may be present at file offset 0x8800.

A.3.2 Analyzing PDF files

O-checker contains `pdfanalysis.py` for analyzing PDF files. This tool scans a PDF file, and it outputs analysis logs describing the final determination of `Malicious!` or `None!`.

Figure 14 shows an example of output from `pdfanalysis.py` against a malicious PDF file with the judgement option (`-j`). The result of the check is shown in Fig. 14 as `Malicious!`. The following describes the tool output:

```
> python pdfanalysis.py -j malware.pdf
00000000-00000008:comment,
00000009-000006E2:obj 1 0 xref from [(8 0 R)]
000006E3-00000721:obj 2 0 xref from [(3 0 R), (8 0 R)]
00000722-0000075E:obj 3 0 xref from [(2 0 R), (4 0 R)]
0000075F-0000083F:obj 4 0 xref from [(3 0 R)]
00000840-000008D9:obj 5 0 xref from [(4 0 R), (6 0 R)]
000008DA-0000090E:obj 6 0 xref from [(5 0 R), (7 0 R)]
0000090F-0000098B:obj 7 0 xref from [(-1 -1 R)]
0000098C-000009DA:obj 8 0 xref from [(7 0 R)]
000009DB-00010E04:obj 17 0 xref from None Suspicious
00010E05-00010E09:xref
00010E0A-00010E28:trailer
00010E29-00010E38:startxref 000039AD
00010E39-00010E3D:EOF,

obj 1 0 xml form
obj 17 0 zlib decompress error
Malicious!
run time: 0.133231163025 sec
```

Figure 14: Example of pdfanalysis.py input and output.

- This PDF file contains nine indirect objects (Nos. 1–8 and 17).
- The indirect object references are listed (e.g., No. 8 refers to No. 1).
- No. 17 is an unreferenced object. (AS7)
- No. 1 is an XML form. (Unrelated to determination)
- No. 17 is a stream requiring the FlateDecode filter, but the decode process for No. 17 fails. (AS8-1)

This PDF file has AS7 and AS8-1 features. This suggests that object No. 17 (located at offset 0x9DB) in this PDF file contains an executable file.

Analyzing encrypted PDF file

O-checker can handle four types of encryption methods, namely 40-bit RC4, 128-bit RC4, 128-bit AES, and 256-bit AES. When the specified PDF file is encrypted, o-checker usually tries to decrypt it using an empty password. If the password is known, you can use the `-p` option to decrypt the PDF file using a specified password.

Exporting objects and streams

You can use the `-o` or `-s` options to export an indirect object from this PDF file for analyzing exploit code.

- The `-o` option decodes the stream of the specified object, and outputs the object in JSON format.
- The `-s` option decodes and outputs the stream of the specified object.

```
> python pdfanalysis.py -j malware.pdf -s 1
<?xml version="1.0" encoding="UTF-8" ?>
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/">
<config xmlns="http://www.xfa.org/schema/xci/1.0/">
<present>
<pdf>
<version>1.65</version>
<interactive>1</interactive>
<linearized>1</linearized>
</pdf>
⋮
<ImageField1 xfa:contentType="image/tif" href="">SUkqADggAACQ
kJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQ
kJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQ
⋮
FQAH/5CQkEOVAAcipwAHuxUAB////5BNFQAHMdcABy8RAAc=</ImageField1>
</topmostSubform>
</xfa:data>
</xfa:datasets>
⋮
```

Figure 15: Example of pdfanalysis.py input and output.

Figure 15 shows an example of the command and its output. In the above, we can find the exploit code in a tiff image.