

FEATURE

SCANNING EMBEDDED OBJECTS IN WORD XML FILES

Christoph Alme

Secure Computing Corporation, Germany

Earlier this year an article by Jan Monsch [1] showcased how rarely-known, ‘alternative’ *Microsoft Word* file formats can be used to transport malware to end users’ PCs. While this doesn’t pose an imminent threat to desktop PCs running an on-access anti-virus scanner, their counterparts running at the network perimeter – for example on web and email gateways – will have to go the rocky road to inspect these alternative formats as well. Even with on-access scanners deployed on corporate end-user PCs, this remains a requirement for gateway anti-virus scanners, since users tend to question their scanners whenever they see their desktop’s scanner block something that the company gateway has allowed through.

Of course, one can argue that activating embedded malware, for example as an OLE object, still needs a significant amount of user interaction. But we can’t rely on that fact to stop it happening, as we simply cannot be sure that end users will not be tricked into activating it through the clever use of social engineering (possibly ‘targeted’ social engineering).

Scanning VBA macros in *Word 2003* XML files has been covered in [2], so this article will focus on the embedding of arbitrary files into *Word 2003* XML files, giving an overview of how they can be found and passed on to the virus scanner. It also shows why this is not such a ‘walk in the park’ as one might at first expect. (If it has the magic ‘XML’ in its name, it ought to be a breeze to parse, oughtn’t it?)

THE ‘WORDML’ XML SCHEMA

All *Office 2003* XML schema files [3] start with an mso-application processing instruction. The actual *Office* application is denoted in its progid attribute, such as ‘Word.Document’ in our case (and only the version-independent ProgID works here).

The WordprocessingML schema’s root element is named wordDocument and since, by default, *Word* defines a ‘w’ namespace for its schema, it is actually <w:wordDocument>. But XML namespaces can be defined with any name – just use xyz, for example:

```
<xyz:wordDocument xmlns:xyz="..." ...
<xyz:docOleData> ...
```

Word will still happily load and render the document correctly. But using a namespace that is deviant from ‘w’ causes a decrease in the number of anti-virus scanners (as hosted on *VirusTotal* [4]) that can detect an embedded,

ZIP-archived EICAR test virus in *Word 2003* XML files from three (as was the case in [1]) to one (at the time of writing this article).

DECODING EMBEDDED OBJECTS

To determine quickly whether a WordML file has any embedded objects at all, the root element’s embeddedObjPresent attribute can be checked for containing ‘yes’:

```
<w:wordDocument ...
  w:embeddedObjPresent="yes" ...>
```

Otherwise, *Word* does not render the document at all in case it does contain embedded objects, and complains.

So this allows a scanner to decide relatively quickly whether it has to parse the whole document at all. When it has to, it should look for <docOleData> elements. Each such element has one child element named <binData>, but can also have any other child element that may simply be ignored by *Word*. Since XML processors are case-sensitive – and the one used by *Word* is no exception here – it may even have a <binData> child element that *Word* will use, next to a <BiNdAtA> child element or similar that *Word* will ignore. Therefore, a construct like this:

```
<w:docOleData>
<w:BinDaTA w:name="oledata.mso">
Bla bla bla
</w:BinDaTA>
<w:binData w:name="oledata.mso">
0M8R4KGxGuEAAAAAAAA
... more base64-encoded data here ...
wMAAAAAAAD/DAAA
</w:binData>
</w:docOleData>
```

could allow scanning of the actual embedded object, contained in the ‘real’ <binData> element, here to be evaded if a scanner only checks the very first <binData> child element of a <docOleData> element without caring for the case of its tag name.

Next, we cannot rely on the <binData> element’s name attribute; it does not need to contain ‘oledata.mso’ for *Word* to treat it as an embedded object – any other name, such as <w:binData w:name="helloWorld.mp3">, will do the job just as well. Using a deviant name allows the results of [1] for *Word 2003* XML files to be decreased to two (at the time of writing this article).

The <binData> element’s data is encoded in base64 (plus intermittent linebreaks). It may contain entity or character references that get resolved by the XML processor. So *Word* will not have any problem rendering a document with a <binData> element’s data encoded, for example, as:

```
<w:binData w:name="oledata.mso">
0M8R4KGxGuEAA&#65;AAA&#65; ...
```

But using such character references here allows the results of [1] for *Word 2003* XML files to be decreased to zero (at the time of writing this article). The same applies when inserting comments into the element data, like

```
<w:binData w:name="oledata.mso">
...
EAA<!-- comment -->AAgAAAAEAAAD// ...
```

Now let's have a look at the actual data. After base64 decoding, we'll get an OLE Compound file with one stream underneath its \Root Entry storage, named as specified in the XML file's associated OLEObject element's ObjectID attribute:

```
<o:OLEObject Type="Embed" ...
  ObjectID="_1218624971" />
```

Note that after base64 decoding, the decoded content is not padded up to the big block size alignment specified in its OLE Compound file header (512 bytes as usual).

The OLE2 stream's data starts with an unsigned 32-bit field denoting the uncompressed size of the following data. The data that follows is compressed using the deflate compression algorithm, but of course we first verify the utilized compression method by checking that the lower nibble of the compressed data's first byte is 8 (= Z_DEFLATED).

After uncompressing, we find yet another OLE Compound Structured Storage file. Seeing the 'D0CF11E0' signature appear once more may remind you of the myth of Sisyphus, but hold on – we can already see some light at the end of the tunnel. If the embedded OLE object supports in-place activation, like a PDF document or *Flash* animation for example, we now find its 'contents' stream directly underneath the \Root Entry and, depending on the actual object's persistence strategy, it may even start with the 'raw' data.

When dealing with an instance of the so-called 'Package' object, which is used to embed arbitrary files and does not therefore support in-place activation, we find its raw data in the '\Ole10Native' stream underneath the \Root Entry. As usual, it is prefixed by a header that consists basically of size fields, a display name and two path names, all in ASCIIZ. At last – we have unveiled the data of interest!

NOTHING IS AS CONSISTENT AS CHANGE

While the *Excel 2003* XML schema does not allow for embedding of OLE objects (or VBA macros), and *PowerPoint 2003* does not have an 'alternative', XML-based file format at all, the upcoming *Microsoft Office 2007* release will change this and more. It will bring a new format called 'Office Open XML', which is supported by *Word*, *Excel*, *PowerPoint* and other *Office* applications. The new file extensions are .DOCX, .XLSX and .PPTX for the default, not 'macro-enabled' document formats, while their 'macro-enabled' counterparts will use the file extensions .DOCM,

.XLSM and .PPTM, respectively. Note that Office Open XML is planned to be the default format, so its prevalence can be expected to increase significantly over the coming years. At the time of writing, *Office 2007* is available as Beta 2 and therefore details may still be subject to change.

Office Open XML files consist of a ZIP archive containing various XML files and the embedded OLE objects as separate archive members called 'oleObject1.bin' and so on. By default, they are located in the 'embeddings' archive folder. But embedded objects don't have to be stored here: a new indirection, called 'relationships', defines where an embedded object's data is stored within the archive. To find out which archive members represent embedded objects, or simply to prevent scanning the whole archive looking for embedded objects, you'll have to start (using *Word* as an example) by parsing the '/word/document.xml' file, looking for <OLEObject> elements (the <wordDocument> element's previously mentioned 'embeddedObjPresent' attribute seems to have vanished):

```
<w:object>
...
<o:OLEObject Type="Embed" ...
  ObjectID="_1218959120"
  r:id="rId5" />
</w:object>
```

The attribute of interest is the relationship identifier, 'r:id'. We can now look up the relationship 'rId5' in the '/word/_rels/document.xml.rels' file to find out where this embedded object's data is stored:

```
<Relationship Id="rId5" ...
  Target="embeddings/oleObject1.bin" />
```

So now we have the path and filename of the archive member comprising the embedded object's data. It appears to be an OLE Compound file containing either a '\Ole10Native' stream or a 'Contents' stream in the usual formats.

To our relief, the outlined embedding approach is consistent among the upcoming *Word*, *Excel* and *PowerPoint* formats – only the names of archive folders, files, XML elements and attributes differ (as of Beta 2, of course). Users of *Office XP/2003* will also be able to use the new formats via the converters already available at [5].

REFERENCES

- [1] http://handlers.sans.org/dwesemann/alternativ_word_formats_v2.0.pdf.
- [2] <http://www.virusbtn.com/pdf/magazine/2003/200302.pdf>.
- [3] <http://www.microsoft.com/office/xml/default.mspx>.
- [4] <http://www.virustotal.com/en/indexf.html>.
- [5] <http://www.microsoft.com/office/preview/beta/convertter.mspx>.