Information Flow in the Peer-Reviewing Process (Extended Abstract)

Michael Backes, Markus Dürmuth, Dominique Unruh
Saarland University
{backes,duermuth,unruh}@cs.uni-sb.de

Abstract

We investigate a new type of information flow in the electronic publishing process. We show that the use of PostScript in this process introduces serious confidentiality issues. In particular, we explain how the reviewer's anonymity in the peer-reviewing process can be compromised by maliciously prepared PostScript documents. A demonstration of this attack is available. We briefly discuss how this attack can be extended to other document formats as well.

1. Introduction

The PostScript language [1] and its successor PDF (Portable Document Format) are the de-facto standards for electronic publishing. Despite PostScript being the older standard, its availability on virtually any platform and its support on a number of printers makes it still widely deployed, in particular in scientific publishing. It is well-known that PostScript is a Turing-complete programming language, and it also comprises commands to access the file system. Some enthusiasts have even implemented a web server [14] or an HTML renderer [8] in PostScript.

In 1992 the community involved in developing the GhostScript interpreter realized that this comprehensive set of commands rises security issues when processing PostScript documents from untrusted sources. This led to the introduction of a configuration option, which prevents the document to access files on the local machine, i.e., to read, write, and delete files. This option is nowadays activated by default. It is commonly believed that this is sufficient to ensure security against malicious documents. (Barring, of course, implementation glitches which tend to appear in and threaten the security of most complex applications.) In the following we argue that this belief is *not justified*. It can be seen as an example of an incomplete modeling of the information flow occurring in the publishing process which in turn gives rise to natural exploits of the weaknesses of the PostScript language.

1.1. The Electronic Publishing Process

To elaborate on our attack, let us first consider the information flow that naturally appears in the electronic publishing process. Usually, the user Alice prepares a document on computer A. Then this document is transferred, e.g., by email, to the computer B of user Bob. Finally, Bob reads the document as rendered by computer B (on screen or by printing it). The information flow of this process is depicted by the solid arrows in Figure 1 (a).

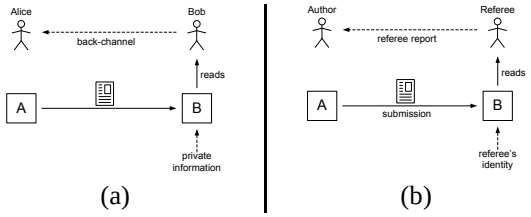

Figure 1. Information flow of electronic publishing, (a) in the general case, (b) in the special case of peer-reviewing.

However, for PostScript documents this description is not complete. Since PostScript is a Turing-complete programming language, the rendered document may depend in arbitrary manner on the data accessible to the PostScript code. This data may—depending on the particular implementation of the PostScript interpreter—contain some of Bob's private information stored on computer B. We will see below that this is indeed the case for common PostScript implementations. In this light, it is necessary to extend the information flow diagram by another arrow (depicted by a dashed arrow below computer B in Figure 1 (a)).

It is this idea of information flow that often underlies—although not explicitly stated—the security considerations concerning PostScript code, and it is this idea that governs the design decisions whether language features have to be disabled or whether they may be available to untrusted documents. In this model, Bob's private information only flows to Bob, but not back to Alice. This is usually considered

to be harmless. As a consequence, in common implementations the access to the private information is not—so we will show—as restricted as it should be.

The flow of information that is usually overlooked in this setting is the human communication between Alice and Bob, in particular, information flowing from Bob to Alice. So a complete diagram should at least contain an additional arrow from Bob to Alice. (In order to make the presentation more concise, we do not include arrows that are not relevant to our discussion.) Including this back-channel, we finally get the situation depicted in Figure 1 (a). Presented in this form, one immediately sees that Bob's private information might in fact flow to Alice.

At this point, one might object that this back-channel is not a security threat since it is not under Alice's control and since the human being Bob will not tell Alice any confidential information even if that information are contained in the rendered document. This, however, is not entirely correct, as Bob may talk to Alice about seemingly harmless information, which may convey the private information through a subliminal channel. So the task of a malicious Alice would be threefold: First, identify some interesting private information available to the PostScript code. Second, devise a dynamic document such that Bob's response will depend on the dynamic details of the document. And third, Alice has to devise an encoding that hides the private information in the dynamic content of the document such that the private information can be reconstructed from Bob's response.

The remainder of this paper is devoted to demonstrating the feasibility and relevance of the above approach. This will be done by analysing and exploiting the information flow in a setting in which PostScript is very popular: the scientific peer-reviewing process.

1.2. The Peer Reviewing Process

The reviewing process in scientific publishing is usually implemented as a peer-reviewing process, the main reason being that reviewing scientific papers requires in-depth knowledge of the subject to judge on its correctness, novelty, and quality. However, judging the work of colleagues potentially bears the danger of decisions being influenced by political considerations, especially if the author who's work is being refereed is aware of the identity of the referee. In particular younger researchers might be afraid to openly contradict established and influential members of the community. For this reason, during a peer-review, the identity of the referee (and often also of the author) is kept secret.

In this light, the identity of the referee, which is usually known to the referee's computer, can be considered as private information whose confidentiality must be ensured. So by applying the considerations of the preceding section to the peer-reviewing process, we see that the information flow is as depicted in Figure 1 (b). It turns out that the identity of the referee is indeed accessible to untrusted documents in many PostScript implementations.

The back-channel is also naturally present in the reviewing process. Since the author usually gets a referee report listing suggestions and mistakes, the author can implement a back-channel by creating a document that dynamically introduces mistakes depending on the identity of the referee. Even if the referee does only report part of these errors, using a suitable error-correcting code one can easily transmit enough information to be able to identify the referee in many situations. So the anonymity of the peer-reviewing process is indeed in danger when using PostScript.

1.3. Related Work

The back-channel we are using can be seen as a novel form of a covert channel [15]. Traditional covert channels typically transmit data on an electronic link, usually a network connection, e.g., by exploiting different execution times resulting in observably different behaviors [17, 3, 9], or by employing steganographic techniques to hide data within other data, see [4] for a survey. In our scenario the PostScript document takes the part of the sender of the data, while the author is the receiver. In contrast to the common setting, the channel we are investigating is not an electronic one but constitutes a socially-engineered back-channel.

Our approach furthermore has similarities to the notion of watermarking schemes [11, 13]. While robust watermarking schemes provide measures that prevent the watermark from being removed from the document, fragile watermarking schemes aim to detect if a document was tampered with. Thus they do not precisely fit our setting as we rather rely on the usual behavior of a reviewer and are not primarily interested in whether an existing document was modified. Another related concept is traitor tracing [10, 6, 5], which allows for detecting a party who leaked a secret, e.g., a secret decryption key. While there are again similarities at the surface, the exact setting as well as the technical realization of our work is quite different since the reviewer does not intentionally leak its secret key but is rather pushed into leaking it without noticing.

Some other surprising PostScript hacks include a Webserver [14] as well as an HTML-renderer [8]. A virus-like program written in LaTeX is described in [16]. Weaknesses of PostScript and the process that led to the current (insufficient) sandboxing model which is implemented on GhostView can be found at the GhostScript homepage [12].

2. Encoding Data in Errors

We will now discuss how the private information can actually be encoded into errors. We will concentrate on the

case of binary errors, i.e., errors that either occur or do not occur. Of course, one could also use errors with higher entropy. As an extreme example, one could insert a random-looking word which is a one-time-pad encryption of the user name or other confidential information. If the referee sends that word back, decoding is easy. In the binary case, the situation is more involved.

We will use errors that are either letter-level or word-level errors (e.g., substituting letters or words by other letters or words), since these seem to be easily detectable. Furthermore, if the errorneous word has the same length as the correct one, these errors can be easily implemented (cf. Section 3). Furthermore, we believe that the errors should be contained in abstract or introduction, since other parts might be read with already lessened concentration. Also, the number of errors should be kept small, and one should avoid errors that are too extreme (like the above-mentioned random word), since otherwise the referee might simply recommend a detailed proof reading instead of listing individual errors.

So in order to encode the username (or whatever information we want to transmit), we first transform the username into a natural number. There are different possibilities how to do this. If the set of potential referees is manageable (e.g., a several dozen or even a few hundreds), one might hard-code the list of referees into the document. Then the username is matched against each referee and the index of the matching referee is used. Of course, the matching routine should be smart enough to match variations of the referee's full name, such as `smith`, `john`, `jsmith`, `johnsmith`, `john smith`, `john.smith`. Such a limited referee list exists in many conferences where the submissions are reviewed by the program committee. (If the document happens to be reviewed by a subreferee, we choose a special index to indicate failure.) If no such list is available, we probably will be able to transmit only part of the username, e.g., the first two letters or the initials, from which the author has to guess the identity of the referee.

When an encoding of the username as a natural number $u \in \{1, \ldots, N\}$ has been fixed, we have to find a suitable code. To encode the username u, this code needs to have N codewords. Assuming that we have n possible distinguishable positions where errors may occur, each codeword has to have length n. Each bit that is set in the codeword corresponds to an error that will occur in the document. Assume that we choose to have w errors in our document. Note that the choice of w is crucial. Too small numbers will make the encoding more difficult, but too large numbers might have the result that the referee will not list individual errors any more. If we fixed a w, this is the Hamming weight of each codeword, so the code is a constant-weight code. Finally, we cannot assume that the referee finds all errors. We should be able to decode given only some of the errors. Let

n	w	e	N
24	4	3	498
24	4	4	10626

n	w	e	N
24	5	3	168
24	5	4	1895

n	w	e	N
24	6	4	532
24	6	5	7078

n	w	e	N
24	7	4	253
24	7	5	1368

n	w	e	N
24	8	4	38
24	8	5	759

Figure 2. Lower bound N on the size $A(n, 2w-2e+2)$ of codes suitable for transmitting data in errors.

e be the number of recognized errors that should be sufficient to decode. This is fulfilled if for any two codewords c_1, c_2, they share at most $e-1$ bits that are set. This holds if and only if at least $w-e+1$ bits are set only in c_1, and $w-e+1$ bits are only set in c_2. And this is equivalent to the fact that c_1 and c_2 have Hamming distance $2w-2e+2$. Let $A(n, d, w)$ denote the size of the largest constant-weight code with codeword length n, minimal Hamming distance d and weight w. Then there is a code satisfying the above conditions if and only if $A(n, 2w-2e+2, w) \geqslant N$. Constant-weight codes are well-studied, e.g., [7, 18] give (constructive) lower bounds for $A(n, d, w)$ for many parameters.

Assume for example, that we have $n = 24$ possible errors. By [7, 18] we get the lower bounds given in Figure 2. Assume that we want to encode the first two letters of the referee name and to add a special index to denote failure. Then $N = 26 \cdot 26 + 1$. Some possible choices for w and e are then $(4, 4)$, $(5, 4)$, $(6, 5)$, $(7, 5)$, and $(8, 5)$. For our online demonstration we have chosen $(w, e) = (8, 5)$, i.e., if the referee detects 5 out of 8 errors, we can decode the two letters. (Note that we have graceful degradation. Even if the referee finds only 4 errors, the set of possible decodings is still quite small.) We have chosen this as a compromise between the number of errors (eight) and the percentage of mistakes the referee may overlook ($36.5\,\%$). Admittedly, eight errors is quite a lot for a single page of text. This can be remedied by either distributing the errors on a longer text or by increasing n. We might prepare a text where almost each word may contain a potential error, resulting in n being in the order of several hundreds.

3. Technical Realization

We now give the technical aspects of the implementation and discuss which interpreters and platforms our methods can be applied to. The results are summarized in Figure 3.

	Environment Variables	Directory Listing	File Access
GhostScript			
– Windows	✓	(✓)[1]	–
– Unix/Linux	✓	✓	–
Adobe Distiller	–	✓	✓
PS Printer	Printer dependent.		

Figure 3. Comparison of the capabilities of some PostScript interpreters

We note that the data accessible to the document varies for different PostScript interpreters. In particular, when the PostScript document is interpreted on a printer, the username often is not available at all. However, at least under Windows the PostScript code is usually interpreted by the computer even when printing, so in this case the username is available. Under Unix, the behavior depends strongly on the printer driver and the capabilities of the printer.

3.1. Identifying the user

The first step for the PostScript document is to determine the username. Depending on the PostScript interpreter on the referee's computer, different methods for reading out the username exist (see also Table 3). GhostScript implements a slightly *extended set of operations* which includes a command `getenv` that allows to read out environment variables. The user name is usually contained in the environment variable `USERNAME` under Windows and `USER` or `LOGNAME` under Linux. GhostScript is by far the most commonly used PostScript interpreter of university employees (note that front-ends like GhostView, GSview, KGhostView, etc. internally invoke GhostScript). Therefore this approach already gives us a fair chance of success.

Another source for acquiring the user name is the *directory structure* of the computer. Note that, while file access is restricted with some interpreters such as GhostScript, directories can be listed on all implementations we are aware of using the command `filenameforall`. This allows to try and detect, e.g., the home directories available on the computer. In case of a single user machine, the user name can be extracted from the name of the home directory. We use this approach in the case of Adobe Distiller, which allows filesystem access but has no command to access environment variables.

At this point it should be noted that Adobe Distiller imposes *no* limitation to file access (as of version 7.0.9). So the complete filesystem can be read and *written*. Since we believe that this imposes a great security threat going far beyond the issue presented in this paper, we have informed Adobe of this problem. They will fix this issue as soon as possible [2]. In our context, this unlimited access of course allows us to retrieve any information from the referee's computer, not limited to the username.

[1] There is an inconsistent behavior of GhostScript under Windows: the command `filenameforall` only lists those directories which have some attribute set (i.e., which are hidden, read-only or system files). The GhostScript source code reveals this to be a bug.

3.2. Introducing dynamic errors

The second challenge is to implement dynamically changing content in PostScript. Since PostScript is Turing-complete, any dynamic changes are possible. However, in practice we do not want to implement a complete typesetting engine in PostScript, but use existing engines like TEX for this purpose. Fortunately, this is possible since TEX allows to include PostScript fragments which are simply passed through into the final document. This PostScript code can then be used to dynamically show or hide parts of the document. So all we have to do is for each error to typeset both the correct and the incorrect spelling at the same place, and to use the PostScript code to hide one of these spellings. Of course, this approach requires that the correct and the incorrect spelling take up approximately the same space.

4. Outlook

We have shown that the fact that PostScript is a programming language can undermine the confidentiality of personal data on the recipient's computer. As an example, we showed how to exploit this weakness to circumvent the anonymity of the reviewer in the peer-reviewing process.

Our result gives rise to several related attacks when using PostScript; we briefly sketch some of them for completeness: (I) Information transmitted back to the originator of the document is not limited to the username. In some cases, malicious PostScript code might, e.g., have access to passwords stored on the hard disk. (II) There is an interesting variant of our attack on the peer-reviewing process that does not even need a back-channel. After identifying the referee, the document could adaptively modify itself to include, e.g., references to the referee's work or comments that are likely to please that particular referee and thus increase the probability of acceptance. (III) In PostScript implementations with unlimited write access (e.g., Adobe Distiller), much more damage can be done since arbitrary code can be installed on the referee's machine. (IV) A contract that changes after having been electronically signed might even have serious legal implications.

Making PostScript resistant to the attack described in this paper, as well as the attacks listed above, is relatively straightforward: If a document cannot gain *any* information

from the computer it is interpreted on, then the attack does not work anymore. This means that the interpreter should provide exactly the same environment to each document it processes, on every computer and on any platform. To the best of our knowledge it is sufficient to disable file access, directorylisting, and environment access. Note that one cannot completely disable these commands, as often parts of the PostScript interpreter are written in PostScript as well. However, since GhostScript already has a mechanism to restrict file access when interpreting user documents, this mechanism could simply be extended. This mechanism could also serve as an example for other interpreters.

Finally, we want to emphasise that PostScript is not the only document format that allows dynamic documents. The Portable Document Format (PDF) allows embedded JavaScript code to change the document. However, we concentrated on PostScript in this document, as it seem to offer the widest range of methods to identify the user, and it allows an easy presentation of the underlying mechanisms.

References

[1] *PostScript Language Reference*. Adobe Systems Incorporated, 1999.

[2] Adobe Systems Incorporated. Personal communication, Nov. 2006.

[3] K. Ahsan and D. Kundur. Practical data hiding in TCP/IP. In *Proceedings of ACM Workshop on Multimedia Security*, 2002.

[4] R. J. Anderson and F. A. P. Petitcolas. On the limits of steganography. *IEEE Journal of Selected Areas in Communications*, 16(4):474–481, 1998.

[5] O. Berkman, M. Parnas, and J. Sgall. Efficient dynamic traitor tracing. In *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 586–595. Society for Industrial and Applied Mathematics, 2000.

[6] D. Boneh and M. Franklin. An efficient public key traitor tracing scheme. In *Proceedings Crypto '99*, volume 1666 of *LNCS*, pages 338–353. Springer, 1999.

[7] A. E. Brouwer, J. B. Shearer, N. J. A. Sloane, and W. D. Smith. A new table of constant weight codes. *IEEE Trans. Info. Theory*, 36:1334–1380, 1990.

[8] T. Burton. HTML renderer in pure PostScript. Online available at `http://www.terryburton.co.uk/htmlrenderer/`.

[9] S. Cabuk, C. Brodley, and C. Shields. IP covert timing channels: Design and detection. In *Proceedings of 11th ACM Conference on Computer and Communication Security*, pages 178–187, 2004.

[10] B. Chor, A. Fiat, and M. Naor. Tracing traitors. In *CRYPTO '94: Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, pages 257–270. Springer-Verlag, 1994.

[11] I. Cox, J. Kilian, T. Leighton, and T. Shamoon. A secure, robust watermark for multimedia. In *Proceedings of Information Hiding*, pages 185–206, 1996.

[12] Ghostscript homepage. Available online at `http://www.cs.wisc.edu/~ghost`.

[13] F. Hartung and M. Kutter. Multimedia watermarking techniques. *Proceedings of the IEEE*, 87(7):1079–1107, 1999.

[14] A. Karlsson. PS-HTTPD. Available at `http://www.godisch.de/debian/pshttpd/`.

[15] B. W. Lampson. A note on the confinement problem. *Communication of the ACM*, 16(10):613–615, 1973.

[16] K. A. McMillan. A platform independent computer virus. Master's thesis, University of Wisconsin-Milwaukee, 1994.

[17] I. Moskowitz and A. R. Miller. Simple timing channels. In *Proceedings of 1994 IEEE Symposium on Security and Privacy*, pages 56–64, 1994.

[18] E. M. Rains and N. J. A. Sloane. Table of constant weight binary codes. Online available at `http://www.research.att.com/~njas/codes/Andw/`.