Scanning Electronic Documents for Personally Identifiable Information

Tuomas Aura
Microsoft Research
Cambridge, UK

Thomas A. Kuhn
Technische Universität München
Munich, Germany

Michael Roe
Microsoft Research
Cambridge, UK

ABSTRACT

Sometimes, it is necessary to remove author names and other personally identifiable information (PII) from documents before publication. We have implemented a novel defensive tool for detecting such data automatically. By using the detection tool, we have learned about where PII may be stored in documents and how it is put there. A key observation is that, contrary to common belief, user and machine identifiers and other metadata are not embedded in documents only by a single piece of software, such as a word processor, but by various tools used at different stages of the document authoring process.

Categories and Subject Descriptors

D.2.5 [**Software engineering**]: Testing and debugging
; I.7.2 [**Document and text processing**]: Document preparation

General Terms

Security, Algorithms, Experimentation

Keywords

Privacy, personally identifiable information, metadata

1. INTRODUCTION

It is well known that digital documents may contain information that is not visible when the document is printed or viewed by the user. Most of the time, the information is there for good reasons. It is needed by authoring and publishing tools to store parameters (e.g., printer settings, author identifiers, etc.) that are not immediately parts of the visible document. It enables different pieces of software in a tool chain to communicate such parameters to each other. Automatically generated metadata also makes it easier to index and search documents in ways that are natural for humans, such as by who created the document and when.

While it is good that computers do things automatically for the user, there is the danger that, if the user is not aware of the presence of metadata or cannot control it, secret or private information may be revealed unintentionally. Sometimes, the information is stored not because it is needed but because it is available. While it is good software engineering practice to leave hooks that may enable future features, the propagation of unnecessary information can be detrimental to user privacy. Information leaks caused by metadata may also violate an organisational security policy.

In this paper, we are mainly concerned with personally identifiable information (PII) and other identifiers stored in a document without the user's knowledge or ability or remove them. Any name, serial number or identifier that pinpoints a unique user, organization, computer or software installation may be used to track the document back to the persons and organisations that created or published it. In this paper, we use the word *publication* in a broad sense to mean either posting the document for public viewing or sending it to selected recipients outside the authors' trusted circle.

We will describe a novel tool for detecting PII in digital documents. The tool is defensive in the sense that it can only be used for looking for offending data in one's own documents. This choice enabled us to make the tool relatively automatic and general. The tool first harvests the user's sensitive identifiers based on various heuristics and then searches for them in given documents in several common encodings. Each document is treated as a flat byte stream that may contain strings at arbitrary locations and in arbitrary encodings. This means that, unlike most PII-detection and removal tools, ours does not need to know where to look. The tool was originally developed to test the PII removal mechanisms in the current and beta versions of Microsoft Office. We report on several case-studies done using our tool. We looked at an ad-hoc collection of documents, at a typical publication process where the document is composed with Microsoft Word and published as PDF, and, finally (mainly for fun) at a collection of anonymized conference submissions.

The PDF case study shows that several widely held beliefs about PII removal are invalid. Firstly, the problems are often attributed to a single piece of authoring software, such as a word processor, and most tools for PII removal are aimed at only one document format. Secondly, it is common wisdom that PII in metadata is removed when a document is converted from the native format of the authoring tools to a print- or display-oriented format such as Postscript, Portable Document Format (PDF) or HTML. We know of at least one major corporation that has a policy of not publishing or emailing word-processing documents; they must be converted to PDF. Unfortunately, we found that unique identifiers are inserted into documents at many stages in the authoring process, and these identifiers often survive format conversions. The identifiers may be added by software components from different vendors who are unaware of each other's metadata, and the components may be combined in ways unexpected by their authors. This means that, in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WPES'06, October 30, 2006, Alexandria, VA, USA.
Copyright 2006 ACM 1-59593-556-8/06/0010 ...$5.00.

order to prevent metadata from appearing in a published document, one must control its insertion and propagation throughout the complete document lifecycle.

The rest of the paper is structured as follows: We first look at some motivating examples and related work. Section 2 explains the architecture of our PII detection tool, including the novel way in which the search strings are harvested and how we deal with variable encodings. The following three sections report the results from our case studies. Finally, section 6 concludes the paper.

1.1 Motivating examples

The goal of this section is to motivate our work by providing examples of real-world situations where users may want to anonymize documents.

Reports and policy documents published by governments usually do not identify the individual non-elected officials that have been involved in their preparation. Anonymity shields the public servants from the pressures of publicity and other undesirable influence. In a recent much-publicised case, the United Kingdom government released a report on the Iraq war that accidentally had the names of the authors left in the metadata [21]. This exposed the authors to undesired public scrutiny.

Commercial companies may have similar requirements for their published documents to represent the entire company rather than the individual contributors. Often, the author names are removed as a matter of general policy. Sometimes, there are specific reasons such as preventing negotiation partners from gaining intelligence on the company's internal decision procedures, or making it more difficult for competitors to hire its key personnel.

When secret military documents are declassified, they are often purged of confidential details including codenames, the names of specific persons, and identifiers that might indicate organisational structure.

One application of anonymity that is probably familiar to the reader is the anonymized review procedure of many scientific conferences. The authors are required to remove their names, affiliations, and obvious references to themselves from the submitted papers in order to allow the referees to assess the merits of the submission without regard to the authors' reputation in the research community. While no strong anonymity is required (it is often possible to guess the authors anyway), the authors may want to remove not only visible occurrences of their names or organisations but also any metadata with similar content.

In some universities, examination papers are anonymized before grading. This reduces the possibility of (unintentional) bias by the graders who may know some students personally. If coursework or examination papers are submitted in electronic format, it might be necessary to remove from the files all data that obviously identifies the student.

The types of information that one might want to remove from documents include the following:

- user names and identifiers
- device names and identifiers,
- organisation names and identifiers, and
- online-services used in the authoring process.

1.2 Background and related work

Early computer security research mostly considered information leaks within operating systems and in the context of multi-level security, where classified information must not leak from a process or user with a high security level to one with a lower level. When information is intentionally released to a lower level, it is downgraded or declassified. Before downgrading, the data can be sanitised. Using this terminology, we can think of the publication of a document as downgrading and purging of the metadata as sanitization. Hidden data in the document forms a kind of covert channel that might bypass sanitization. (For an introduction to computer security models, see, for example, [4].)

Apart from compliance with security policies in the presence of adversaries, the user may be worried about accidental disclosure and privacy. Privacy is a broad concept with many definitions. In the context of this paper, it means that users want to know and control where their personal information is stored and propagated. The user may wish to remain anonymous or pseudonymous [19]. Unwanted metadata in digital documents can cause violations of both organisational security policy and user privacy.

The information leaks related to digital documents that have received most publicity have not been caused by PII or metadata but by inadequate attempts to redact secret documents before publication. For example, in 2000, the New York Times published on the web a secret CIA document about a coup in Iran [20]. The newspaper intended to erase the names of the persons involved but did this by painting white rectangles over them with a PDF editor. Obviously, the names were left in the document and could be recovered easily. In a similar case, the Washington Post blacked out a credit card number and other details when it published a ransom letter from the Washington sniper in 2002 [9]. The U.S. Department of Defense has made the same mistake: in a recently published report on a checkpoint shooting in Iraq, parts had been censored by covering them with black rectangles in the PDF [12].

Documents created with old versions of Microsoft Office have also been known to retain fragments of deleted data in the binary file. Considering our discussion of embedded objects in section 4.3, the most interesting case is that of embedded OLE objects that contained fragments of deleted data from the source document [15]. Murdoch and Dornseif [17] discovered another common channel for information leaks: thumbnails embedded in digital-image files. These small images are intended for preview and should be identical to the main image. Unfortunately, some image editing software does not update the thumbnail when the main image is modified. This means that the thumbnail contains an uncropped or unedited version of the image.

An early example of unique identifiers in digital documents is the GUIDs that were used in Microsoft Office documents prior to the year 2000. They included the hardware address of a network interface on the author's computer to guarantee global uniqueness [13]. (In later versions of the software, the GUIDs are generated entirely randomly and we did not encounter the old type identifiers in this work.) The privacy concerns about the GUIDs and other unique identifiers, such as those of wireless and embedded devices, microprocessors and digital media, are summarised well by Markoff [14].

Apart from forbidding the use of specific document formats, the first solution to the discovery of offending data items is to provide instructions for removing them (see, e.g., [16]). There are also many software tools for cleaning documents of metadata. Finally, the current and future versions of Micrsoft Office provide built-in features for removing any metadata. The common weakness of the metadata-removal mechanisms is that they only remove data from known locations in the document.

In most cases reported in the literature, the offending information has been found in an ad-hoc manner, by searching or browsing the document contents with a tool other than the software with which

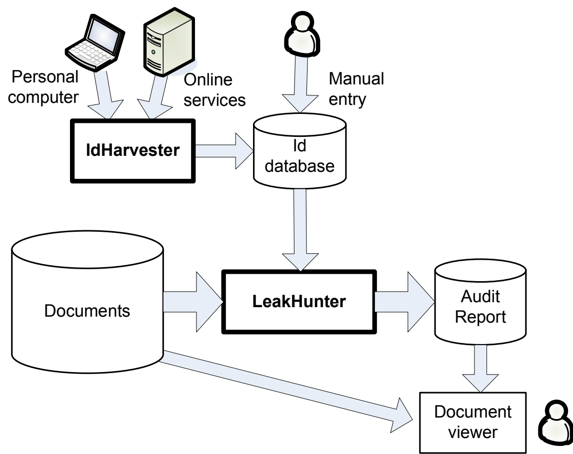


Figure 1: PII detection tool architecture

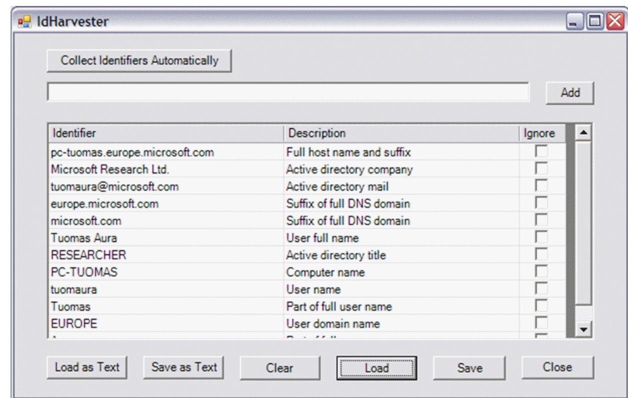


Figure 2: IdHarvester automatically collects search strings

it was created. Even when there is a reason to think that the information has been found by a systematic search, the goal of the adversary has been to find a few pieces of embarrassing data rather than to find and remove them all. Byers [6] presents the results of a more systematic search for hidden data in Word documents. In particular, he compares the strings in the file to the ones visible in the document. The advantage of this approach is that it can detect data in locations that the user and the tool creator are not aware of. Our tool gains an additional advantage from the fact that it is purely defensive and knows which strings to search for. This enables us to consider various data encodings that might not be easy to detect with a simple string-extraction tool.

One preventive approach to information leaks is *tainting*, i.e., tracking the propagation of private data that must not be sent outside a confined system. The idea originates from a short-lived experiment in Javascript 1.1 [18]. Chow et al. [7] apply tainting to track the propagation of sensitive data in web servers. Similarly, one could mark PII and any data values derived from it as tainted and prevent the sending of tainted data out of the system. This approach is probably too inefficient to work in a production desktop system but could be used to detect information leaks during software testing.

2. PII DETECTION TOOL

This section describes the tools that we developed for detecting PII in digital documents. More specifically, the purpose of the tools is to detect identifiers in documents that could be used to trace a document back to the persons, machines, services and organisations that were involved in authoring the document. While there are many ways to approach such a task, we made the following design decisions:

- We want to detect identifiers in locations that are previously unknown to us. Indeed, the first reason to build the tool was to discover where PII is hidden in documents and to audit the results of software that promises to remove metadata. This constraint leads us to look at string search for known identifiers. It also means that we could not use any of the many metadata removal tools on the market because they only find data in known places in the document.

- The searching should be as automatic as possible. In particular, it should not require the user to manually enter the search strings. How this was achieved will be explained in section 2.1.

- The tool should be able to cope with various character encodings, including more than one layer of encoding. Nevertheless, it should have acceptable performance even on large document sets. The challenges and the trade-off that we found will be described in section 2.2.

Figure 1 shows the tool architecture. The main component is the LeakHunter that searches for a given set of strings from a collection of documents. The set of search strings is provided by another component called IdHarvester. The output of the search is a report of the matched identifiers and their context.

2.1 Automatic harvesting of search strings

Our tool needs to know which strings to look for. We do not, however, want the user to manually enter the search strings. Instead, we provide an automated mechanism for collecting potentially compromising strings from the user's computer and from the Active Directory (AD), which is an administrative database in a Windows domain and contains various pieces of PII and organisational data. The collected data items include, for example:

- user's real name, username, domain, security id
- computer's NetBIOS name, domain, DNS name and suffix
- user's email addresses, mailing addresses and telephone numbers
- organization name
- names and addresses of various online servers such as domain controllers, email and webmail servers, instant messaging servers, file and document servers, print servers and printers

The list of places to obtain these identifiers is built into the tool and can be easily extended. Although we have kept adding new heuristics for finding and deriving potentially offending identifiers, most information leaks in real documents are of the obvious ones like the username.

These strings are broken up into substrings based on spaces and other delimiters. For example, the name “John Smith” will be converted to three search strings: “John”, “Smith” and “John Smith”.

Internet domain names are treated in a similar way, so “europe.microsoft.com” creates additional strings “europe”, “microsoft” and “microsoft.com”. (The top-level domain “com” is excluded).

The user is also allowed to enter new strings manually or to import them from a file. Figure 2 shows a screenshot of the ID Harvester.

2.2 Searching various data encodings

The LeakHunter search engine is designed to work even when the document encoding is unknown. Thus, it must try several possibilities. The main challenge is to find an appropriate balance between efficiency and supporting a large number of string and character encodings. We have implemented the following:

- upper and lower-case characters
- 8-bit ASCII characters, Unicode UTF-16 (little and big endian), UTF-8
- URL %-escapes, XML entity references and character references [2], C string escapes
- replacing accented characters with unaccented and vice versa
- common character variations, such as “ä” written as “ae”
- replacing whitespace with other whitespace
- binary strings (e.g., IP and MAC addresses)
- NetBIOS machine names [1]

Other encodings that should be supported in the future are ISO-8859 encodings, EBCDIC, and internationalised domain names [8].

The four main difficulties that we encountered were non-unique, variable-length, unaligned and layered encodings. By non-unique, we mean that the same string can be encoded in many different ways. In the worst case, the number of encodings may be infinite. The simplest example is that the search should work for both upper and lower-case strings. When searching for “John Smith”, we ought to match also “JOHN SMITH”, “john smith”, “john SMith” etc. Even larger numbers of combinations are caused by escape sequences, such as the URL encoding. “John Smith” ought to match “John%20Smith” and even strings like “J%6fhn%20Smit%68”.

Another problem is created by variable-length character encodings, such as UTF-8. It is impossible to know the length of an encoded string without decoding each character separately. This makes it impossible to use some efficient string-search algorithms, such as Boyer-Moore [5], that aim to skip strings without making any comparisons. The obvious solution is to encode the search string first in UTF-8 and then search for it as binary data. This is possible if there are only a few different encodings to try. The same solution does not work for non-unique variable-length encodings such as the escape sequences demonstrated above because there are too may combinations to search for.

Since we do not want to assume anything about the document’s encoding and do not try to guess it based in the document type, we cannot assume that the strings or characters are aligned to any specific word boundaries. For example, in the UTF-16 encoding, each character is represented by two bytes. The document might consist of UTF-16 text preceded by a header in a different format, and the header might contain an odd or even number of bytes. Thus, we need to match UTF-16 characters aligned to both odd and even byte boundaries. Moreover, short UTF-16-encoded strings may appear at unpredictable offsets anywhere in the document. The same problem applies to any multi-byte character set.

Layered encoding means that the text has been encoded first in one way and then in another way. This is common, for example, when URLs are stored as Unicode strings and contain URL %-escapes. When we remember that both the URL and the A–F digits in the hexadecimal escape sequence may be in upper and lower case, there are already four layers of encoding. Ideally, we should support any number of encoding layers and either arbitrary combinations or all sensible combinations of encodings. That task would, however, be equivalent to parsing the document with a context-free grammar, which has a worst-case complexity of $O(n^3)$ where n is the length of the document. Although parsing is faster in practice, it would still be too slow for large sets of documents (long text) and encodings (large grammar).

In order to find an acceptable compromise, we experimented with two types of search algorithms, described in the following two sections, which both have their advantages and limitations.

2.3 Fast string-search algorithms

For a single search string, the fastest known algorithm is Boyer-Moore [5]. The Aho-Corasick algorithm [3], which uses a a prefix-tree-like automaton, is more efficient for large sets of search strings and, thus, faster in most of our experiments. These algorithms look for exact matches of one or more search strings in the text. It is trivial to modify these algorithms to match both upper and lower-case characters but, in general, handling a large number of encodings is problematic. In the general case, we create all possible encodings of the search string using a particular encoding method, and then consider each encoding as a different search string. The resulting search can be very fast but slows down (almost) linearly with the number of encoding combinations. In practice, we can search for each string in ASCII format and in the various Unicode encodings, and match characters case-insensitively. It would not be possible to support the almost infinite number of combinations caused by various escape sequences.

2.4 Regular-expression matching

Regular expressions provide a more compact way of expressing the combinations created by non-unique and layered encodings. Figure 3 shows how each character in the search strings is expanded into a regular expression. For each layer of encoding, the alphabet in the previous regular expression is replaced with sub-expressions. The regular expression length grows exponentially with the number of encoding layers, which is why we only show 4 layers in the figure.

The search strings are expanded into regular expressions by concatenating the regular expressions for each character. For a set of strings, we structure the expressions in the form of the prefix tree. Depending on the number and length of search strings, and on the number of encoding layers, the resulting regular expression can be thousands or even millions of characters long. This is quite different from the more typical use of regular expressions, where a relatively short expression is typed in by a human.

The regular expression is converted into a nondeterministic finite automaton (NFA). The fastest search algorithms further compile the NFA into a deterministic finite automaton (DFA). The latter step is, however, impossible for us because the size of the DFA may grow exponentially with the size of the NFA (and the size of the regular expression). Computer memory is clearly the limiting factor in this type of regular expression search but, as long as the NFA fits into the memory, the search happens at acceptable speed. We implemented our own regular expression library to cope with large expressions and because of the fact that we ultimately want to search binary data, not text. We did not, however, find any partic-

<table>
<tr><td>original character
k</td></tr>
<tr><td>upper and lower case
(K|k)</td></tr>
<tr><td>URL %-escapes
((%4b)|(%6b)|K|k)</td></tr>
<tr><td>upper and lower case
((%4(B|b))|(%6(B|b))|K|k)</td></tr>
<tr><td>ASCII, UTF-8, little and big-Endian UTF-16
(((\x25|(\x25\x00)|(\x00\x25))(\x34|(\x34
\x00)|(\x00\x34))(\x42|(\x42\x00)|(\x00\x42)
|\x62|(\x62\x00)|(\x00\x62)))|((\x25|(\x25
\x00)|(\x00\x25))(\x36|(\x36\x00)|(\x00
\x36))(\x42|(\x42\x00)|(\x00\x42)|\x62|
(\x62\x00)|(\x00\x62)))|\x4b|(\x4b\x00)|
(\x00\x4b)|\x6b|(\x6b\x00)|(\x00\x6b))</td></tr>
</table>

Figure 3: Regular expressions for encodings of 'k'

ular techniques to handle huge regular expressions and very little about them is said in the literature.

The theoretical limitation of regular expressions in comparison to context-free grammars is that the number and order of the encoding layers has to be fixed. In practice, we can cope with upper and lower case characters; replacing whitespace with other whitespace; replacing accented characters with unaccented and vice versa, other common character variations; the URL, XML and C escape sequences in upper and lower case; and the ASCII and Unicode character encodings. Making the different types of escapes alternative rather than layered on top of each other (i.e., combining them into one layer) leaves us some scope for adding new encoding layers. Otherwise, with the current algorithms, it is not possible to add further layers without a significant performance hit caused by excessive memory consumption. Typically, the creation of the NFA and the search take minutes. The advantage compared to the exact string-search algorithms is that a large number of encodings and combinations of encodings is exhaustively considered.

We do not currently support decryption or decompression of data. When the data is encrypted, the encyption usually prevents data leaks anyway. We are only trying to detect accidental disclosures of PII; we are not trying to detect malware that uses encryption to covertly leak PII. Support for compressed documents, on the other hand, is left for future work. Similar to compression are mechanisms that encode binary data as ASCII, such as the Base64 encoding commonly used for email attachments [10]. Although otherwise not difficult to decode, strings in this encoding do not always start at a byte boundary.

2.5 False negatives, positives and other limitations

It is difficult to know how many false negatives there are, i.e., how many privacy-compromising strings are missed by the search. This is because we can only search for the kinds of identifiers and encodings that we know about. Clearly, this type of tool can never be guaranteed to find all unwanted data in documents. It can, however, be argued that our tool looks for many more identifiers than a user would consider in an ad-hoc search, and that the regular expression search covers exhaustively all combinations of string encodings composed of the supported layers. Thus, our tool provides significant additional assurance compared to previous document inspection techniques. It complements other tools and practices and can find information leaks that we otherwise would be unaware of.

The lack of guarantees means that our tool is not suitable for all applications. For example, it is not suitable for redacting classified documents, although it can be used for auditing the results. In the current form, it cannot be used for anonymizing medical case reports because the patient is not the author of the medical documents. That is, we only address situations where the author is the person who requires anonymity. Often, the style and contents of the document may be sufficient to recognise the author of a document, such as a novel or a political declaration. We obviously cannot detect such leaks. In legal documents, professional interpretation is required to determine what must and what must not be disclosed, and no automated tool can make the decisions. Sometimes, there is no need for sanitization. For example, when the data is entered via a simple text interface like a web form, it is easy to avoid saving any metadata. Currently, we have only considered offline detection of PII and the tool is not suitable for online detection in networks. With further development it may, however, be possible to integrate leak detection to an application-layer firewall, document server, or email gateway.

False positives are also a major challenge to the usability of the search tool. The main techniques for dealing with them are based on heuristics that sometimes require user intervention. We exclude search strings that are known to cause problems, such as the "com" or "net" at the end of DNS names. The most problematic identifier in our tests was "microsoft", which is not only the current authors' organization but also appears in the name of many common authoring tools and, thus, could be found in almost all files. This is, fortunately, a special case that does not apply to all users. Very short strings (e.g. the "TU" from "TU München") cause many false positives because they are common parts of English text or are likely to occur by chance in binary data. To avoid this problem, we usually set a minimum length of three characters for search strings and allow the user to fine tune the set of search strings to include or exclude short names and acronyms.

Another way to deal with false positives is to present the search results in such a way that the user can easily see the context in which the identifiers were found and ignore any uninteresting cases. We currently provide hexadecimal and text dumps of the data surrounding the match.

3. WHERE PII IS HIDDEN

We first looked at a collection of miscellaneous documents found on our own computers. In this section, we list various locations in which metadata was found. Although we afterwards found references to all these types of metadata in the literature or software documentation, some were surprises to us.

Human-readable metadata. Office tools, including word processing software, include features for entering metadata into documents. This metadata typically includes the document title, author, author's affiliation, keywords, etc. The metadata may be generated automatically or added by the user. It is used mainly for searching and indexing documents, and to store information about the document's history that may be helpful to the user. Authoring software typically allows the user to scrub this kind of explicit metadata from the document.

Machine-readable metadata. Authoring software may store information about the history of the document automatically. This information is needed mainly to remember user choices, such as print or style settings. It is also used to facilitate collaboration, for example, by correlating revisions of the same document. Un-

expectedly, we found little evidence that these types of metadata cause any real harm to privacy. The reasons are that the metadata is often stored outside the document, in the file system, Windows registry or an application-specific database, and that the GUIDs used to identify documents in Windows are now random numbers (see sec. 1.2).

Easily ignored printable data items. While the authors tend to carefully review the main text of the document, some additional parts, such as the page header and footer, may be accidentally ignored. Some of these data items may be printed only in special situations, such as when producing handouts for a presentation. Fortunately, authoring software increasingly warns users about this type of data.

Tracked changes. Judging by anecdotal evidence, the type of hidden data that has caused most embarrassment is tracked changes or undo information (see sec. 1.1). Such information is needed while the document is being edited but should obviously be deleted before publication. The problem is that authoring tools have not always made much difference between saving a document for continued editing or for publication.

Human-created comments. Documents often contain comments that are not printed with the rest of the text. For example, presentations are accompanied by a script for the speaker, which is hidden from the audience. We found that fractions of the speaker's scripts were sometimes left in place when slides were cut and pasted from one presentation to another or when one presentation was used as a template for others. Word-processing documents may also include comments that are not printed.

Machine-generated comments. Machine-readable parts of a document may also contain comments. Their purpose may be to facilitate debugging, help product or add-on development, or to store metadata that is not a part of the original document format. For example, Postscript files routinely contain the name of the software that was used to create the document, the creation date, and the file name of the source document. Sometimes, they contain the username of the document creator. HTML files also tend to be full of comments that may reveal unwanted facts about the documents' history.

Hyperlinks. Hyperlinks are addresses of other documents such as web pages. The problem with them is that the addresses to which they point are typically not visible when the document is printed. This means that they may be ignored when the document is reviewed for publication. Yet, the addresses in unintentionally retained links may refer to web sites within the author's organization.

Metadata in embedded objects. Typical documents have not been created with a single piece of software but contain embedded tables, figures and other types of objects. The tool with which the main document is created, such as a word processor, often has no knowledge of the format or contents of the objects. Instead, the editing and printing of the embedded objects is delegated to other pieces of software, possibly from different vendors. One type of embedded object that deserves special attention is digital images. Images often contain metadata such as the date on which the picture was taken, the model of the camera, exposure information, the author name, and a thumbprint image for preview. (We will have more to say about embedded objects in section 4.3.)

Names and paths of embedded or linked objects. Embedded objects are often linked to their source documents, for example, for the purpose of keeping the object up to date with any changes made to the source. These links are similar to hyperlinks except that they typically point to a file on the local disk rather than to the web. The link reveals the name and path of the file from which the embedded item was copied. The path often contains the username of the document author.

Template and style names. Similar to embedded objects, documents contain links to templates, style sheets (CSS) and other types of external style information such as background images and sounds.

Macros and scripts. Documents may contain or link to macros and scripts, which are essentially program code. The code includes comments, variable names and other programming idioms that reveal more information than is necessary for the functionality that it implements. Moreover, macros are sometimes collected in document templates just in case they might be needed. Macros are usually associated with Microsoft Word and Excel but other software packages have similar features. For example, PDF may contain Javascript and Emacs allow macros to be stored at the end of every file. (Indeed, the latter feature was used in typesetting this paper.)

4. COMPLETE AUTHORING PROCESS

As a case study, we looked at a typical authoring process that starts from a word processor or text editor and ends in a published PDF document. This example was chosen partly because it is common wisdom that documents should be published in PDF format rather than as word processing documents, such as those produced by Microsoft Word. The main observations are summarised in the following sections.

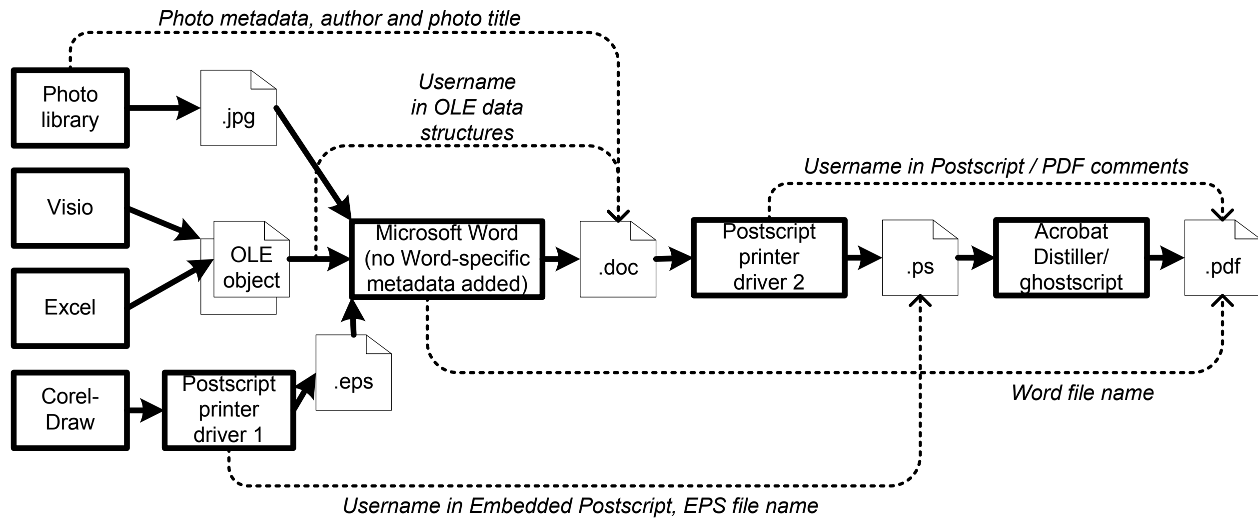
The important insight that we gained from this study is that the process in which metadata is added to documents is far more complex than is usually thought. It involves numerous software components from various vendors that may all add their own pieces of data into the document. The document authoring process depends on a tool chain that is used for creating and transforming components of the document, which are then compiled into one publishable entity. This final document may again be transformed by several tools. The final document may be used as a component in future editions, which means that the document lifecycle does not end at the publication.

Figure 4 shows the particular processes that we considered in this study and the kinds of metadata that was found. The dotted arrows show the lifespan of the data from where they are inserted to the last file in which they appear. Figure 4(a) shows an intentionally constructed worst-case scenario with Microsoft Word 2003 as the main authoring software. Note that we used the Word feature for removing PII. Figure 4(b) shows a typical process for scientific publishing with LaTeX. It should be noted that the two scenarios are not comparable; we could have constructed the same kind of worst-case scenario for LaTeX as for Word. Instead, we chose to demonstrate that even the minimal process has some leaks.

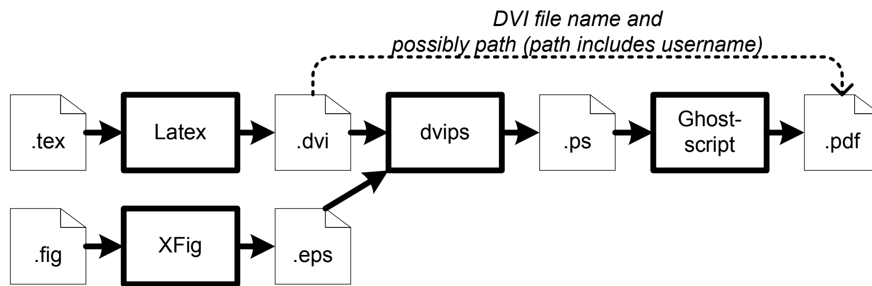
4.1 Printer driver

The first major observation made in this case study was that, even if the authoring software carefully removes all metadata from the document, Postscript printer drivers may unintentionally put it back.

Printer drivers usually send to the printer metadata in addition to



(a)



(b)

Figure 4: Two examples of a PDF authoring process

the visible contents of the document. This data typically contains the name or username of the user who invoked the printing function. The printer needs to know the name, for example, to make it easier to browse and delete print jobs from the printer user interface. Since this feature is well understood by users, it may seem unlikely that any privacy issues would arise from such practice. Postscript printer drivers are, however, commonly used outside the original context for which they were developed. The drivers are used for converting documents into Postscript files which are then published electronically, rather than being sent to a local printer. Often, the physical printer is not even present on the system. Although not intended for that purpose, Postscript is used as a document interchange format.

Postscript contains the same metadata regardless of whether it is sent to the printer or saved as a file. For example, the following headers (i.e., comments) are from Postscript files created in our tests using different printer drivers.

```
%%Title: Microsoft Word - Testing.docx
%%CreationDate: 1/23/2006 19:30:21
%%For: tuomaura

%%OID_ATT_JOB_OWNER "tuomaura";
%%OID_ATT_JOB_NAME "Microsoft Word -
  Testing.docx";

%%Creator: CorelDRAW 10
%%Title: test-figures.ps
%%CreationDate: Thu Apr 14 14:32:47 2005
%%For: Michael Roe
```

As can be seen above, the Postscript files contain the original file name and the name or username of the person that created them. Since the Postscript headers are manufacturer-specific, all printer drivers produce slightly different comments. The alarming observation is that this metadata appears in the Postscript file even if all metadata is carefully purged from the original document. It should be noted that different Postscript printer drivers behave differently and it is possible to find ones that do not store any metadata.

Clearly, the authoring software does not, and cannot, know about the metadata added by the printer driver. The printer manufacturer, on the other hand, has no reason to consider printing to a file as a significant application of the driver. This kind of loose coupling between the software components means that the ultimate responsibility for managing metadata is left to the end-user or organization.

4.2 PDF conversion

While Postscript can still be used for online publishing of printable documents, it has been mostly replaced by the Portable Document Format (PDF). We tested multiple methods for converting documents to PDF. This is often done by printing the document into a Postscript file and then using special conversion software to turn the Postscript into PDF. We tested two common conversion tools: Adobe Acrobat Distiller and Ghostscript. By default, both copy the metadata from Postscript comments into the PDF file (although for Ghostscript this appears to be version dependent). For example:

```
/Title(Microsoft Word - Testing.docx)
/Author(tuomaura)
```

Thus, the metadata from printer drivers may be propagated to the final PDF file. The conclusion is that publishing documents as PDF does not guarantee freedom from unwanted metadata unless the creator has complete control over every stage in the conversion process or cleans the final document of metadata with a PDF editor. In general, when the final document is produced with a tool chain that includes components from multiple vendors, such as a printer manufacturer and different software publishers, metadata may be introduced at any stage in the process. It is necessary but not sufficient for privacy to be able to clean out metadata in the main authoring tool such as the word processor.

One solution would obviously be a more integrated process for producing the final document. For example, XFig avoids the pitfalls associated with printer drivers because it has built-in support for Postscript output. For another example, we experimented with a save-as-PDF feature in a word processor and did not find it to insert any unwanted metadata. While this may be the best solution for the particular problem of PDF conversion, it is clear that there will always be situations where it is desirable to use a chain of independent tools and software components. In such cases, it is important to consider the privacy implications.

In the LaTeX-based process of Figure 4(b), similar problems occurred in document conversion. Depending on how it is in invoked, the *dvips* conversion tool adds either the DVI file name or its full name and path to the Postscript output. The file path usually contains the username. This information is propagated all the way to the PDF thought Ghostscript.

4.3 Embedded objects

Complex documents often contain embedded objects that have been produced with a different authoring tool. The objects may contain metadata that is not detected and cannot be removed by the software that processes the main document. In this study, we tested three types of embedded content: Embedded Postscript (EPS) figures, JPEG digital photographs and Object Linking and Embedding (OLE) objects, all of which were added to Word documents. All three kinds of embedded content were found to hide some metadata.

Embedded Postscript objects, just as any other Postscript documents, can contain headers and comments. When the main document is printed, the EPS is simply copied into the output file. Thus, any metadata in the EPS file will also be in the output Postscript. The embedded headers are not converted to PDF metadata, though, because those values are taken from the main document.

Digital images contain metadata inside the image file in the Exif format [11]. Most of the data is put there by the digital camera, such as the date when the picture was taken and exposure information. Photographers may add other notes such as the name of the artist. When the JPEG file is embedded in a Word document, all this metadata is retained. When printing the main document to a Postscript file, the information is lost, however.

OLE objects can be produced with any software that is compatible with the specification. If the main document and the embedded object are produced by software from different vendors, it is generally impossible for one to know about the metadata in the other. Thus, the data must be purged separately from each object and from the main document. There is clearly a need for a standard interface for propagating the instructions for metadata-deletion to the objects. Moreover, the data structures for embedding objects may contain information that exists neither in the original main document nor in the embedded object. This information includes the file path and name of the source file of the object and the username of the person who did the embedding. The file path is needed as long as the object is linked to the original file while the username is clearly unnecessary information. It is possible to break the link to the object file and, thus, remove the file information. Future versions of Word also avoid storing the username in the OLE data structures.

5. ANONYMOUS CONFERENCE SUBMISSIONS

Out of curiosity, we used the PII detection tool on anonymized conference submissions from two computer-security conferences where privacy was a topic of interest. Clearly, it is a problem that we need to know which strings to look for. Since we did not want to breach the anonymity of the submissions, the test was done after the publication of the conference program. As search strings, we used the names, affiliations and email addresses (which often contain the username) of the authors of the accepted papers. We searched for these strings in the original, supposedly anonymous, submissions in Postscript and PDF formats. Any matches in the list of references or other printable text were ignored as such occurrences are typically intentional.

We found that 3 out of 43 submissions had not been anonymized correctly. One PDF document contained the authors' names in the `\Author` field, one Postscript file contained the author's name inside an EPS figure, and another one contained the author's username and name in an EPS figure. Another Postscript submission was found to contain the file path of the source DVI file, including the author's username, but our tool did not detect it because it was not given the right search string.

This sample is by no means representative of all conference submissions and the number of mistakes found was, in fact, lower than expected. As the anonymization is only intended to protect against inadvertent bias (not a malicious attacker), authors may not have been as careful to anonymize their submissions as they would have been in a different situation. Therefore, we will not try to draw conclusions based on the exact number of incorrectly anonymized papers. However, it does show that when asked to produce an anonymous electronic document, even security experts don't always get it right. In normal operation, a tool like ours would be run by the submissions web site or, preferably, by the authors. In the former case, the search strings could be obtained from the submission form while, in the latter, they could be collected from the author's computer.

6. CONCLUSION

We described a tool for auditing documents for PII, including unintentionally hidden identifiers, that could be used to trace the document back to the authors or their organisation. The tool has a novel feature for automatically collecting potentially offending identifiers and it searches for them in various encodings anywhere in the document without needing to know anything about the document structure. We used the tool to discover where and why user, machine and organisation identifiers are left in published documents. While most of the literature focuses on the shortcomings of single pieces of software, we found that the modern document authoring process typically involves a chain of tools and software components, such as format converters and printer drivers, that are often used in ways not envisioned by their developers. Each software tool or component may contribute to the metadata in the document. Therefore, in order to create documents without identity-revealing data, one must carefully consider every part of the authoring process.

7. REFERENCES

[1] Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods. RFC 1001, March 1987.

[2] Extensible markup language (XML) 1.0 (fourth edition). W3C recommendation, August 2006.

[3] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[4] Matt Bishop. *Introduction to Computer Security*. Addison Wesley, 2005.

[5] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

[6] Simon Byers. Information leakage caused by hidden data in published documents. *IEEE Security & Privacy Magazine*, 2(2):23–27, March/April 2004.

[7] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proc. 13th USENIX Security Symposium*, pages 321–336, San Diego, CA USA, August 2004.

[8] A. Costello. Punycode: A bootstring encoding of Unicode for internationalized domain names in applications (IDNA). RFC 3492, 2003.

[9] Kurt Foss. Washington Post's scanned-to-PDF sniper letter more revealing than intended. *Planet PDF*, October 2002.

[10] N. Freed and N. Borenstein. Multipurpose Internet mail extensions (MIME) part one: Format of Internet message bodies. RFC 2045, November 1996.

[11] Japan Electronics and Information Technology Industries Association. *Exchangeable image file format for digital still cameras: Exif Version 2.2*, April 2002.

[12] Anick Jesdanun. Data leak highlights complexities of electronic documents. *Associated Press*, 4 May 2005.

[13] Paul J. Leach and Rich Salz. UUIDs and GUIDs. Internet Draft draft-leach-uuids-guids-01, IETF, February 1998. Archived at http://www.watersprings.org/.

[14] John Markoff. A growing compatability issue in the digital age: Computers and their user's privacy. *New York Times*, March 1999.

[15] OLE update for Windows 95. Knowledge-base article 139432 revision 2.2, Microsoft, August 2004.

[16] How to minimize metadata in Word 2003. Knowledge-base article 825576 revision 2.6, Microsoft, January 2006.

[17] Steven J. Murdoch and Maximillian Dornseif. Far more than you ever wanted to tell: Hidden data in Internet published documents. Presentation at 21st Chaos Communication Congress, December 2004.

[18] Netscape Communications Corporation. *Client-Side JavaScript Reference*, May 1999.

[19] Andreas Pfitzmann and Marit Hansen. Anonymity, unlinkability, unobservability, pseudonymity, and identity management – a consolidated proposal for terminology. Technical Report v0.26, TU Dresden, December 2005.

[20] James Risen. Secrets of history: The C.I.A. in Iran. *New York Times on the Web*, 2000.

[21] Richard M. Smith. Microsoft Word bytes Tony Blair in the butt. Web posting, computerbytesman.com, June 2003.