Can Content-Based Data Loss Prevention Solutions Prevent Data Leakage in Web Traffic?

David Gugelmann | ETH Zurich
Pascal Studerus | AdNovum Informatik AG
Vincent Lenders | armasuisse
Bernhard Ager | ETH Zurich

The effectiveness of data loss prevention (DLP) solutions remains obscure, especially with respect to the types of attacks they claim to protect against. A systematic analysis of HTTP data leakage vectors and a test of three content-based DLP solutions give insight into these solutions' vulnerabilities.

Data leakage is a serious threat to many organizations. A recent study of 350 companies estimated that the average total cost of a data breach incident is US$3.8 million.[1] To address this threat, companies have deployed perimeter protection mechanisms such as firewalls to control and block undesired dataflows. However, attackers have quickly adapted their data leakage techniques to evade these protection schemes. For example, today's malware often relies on HTTP as a command-and-control channel to hide its activities in the regular flow of Web traffic.[2] Because blocking HTTP traffic at the perimeter would prevent users from accessing the Web, this communication channel is often left open. Therefore, perimeter security alone is no longer sufficient to address modern data leakage threats, and *defense-in-depth* concepts are required. This means that organizations expect weaknesses in their protection measures and employ layered defenses that prevent attacks as well as incorporate attack detection and mitigation measures.[3]

Data loss prevention (DLP) solutions—systems that analyze dataflows and enforce data policies to safeguard sensitive information—promise to help. Due to the variety of technical approaches that can address data leakage threats, it's not surprising that many DLP solutions are currently on the market.[4,5] More than a dozen security companies promote their solutions as a silver bullet against leakage. However, selecting a proper solution isn't a trivial task as there is little transparency in this area: vendors are generally reluctant to reveal how their systems work in detail and provide only high-level descriptions in their product documentation. In addition, threat models aren't defined explicitly, so the DLP solutions' effectiveness against particular threats remains unjustified.

In an attempt to fill this gap, we provide a systematic evaluation of content-based DLPs to examine whether they effectively prevent data leakage in Web traffic. Although the DLP solutions we tested received very high marks in other evaluations, our tests demonstrate that these solutions protect against accidental data loss but can be bypassed by attackers. We specifically focus on HTTP, as it's the dominant Internet protocol and many organizations allow Web browsing.

Data Leaks over HTTP

HTTP is a stateless request–response protocol that browsers use to access the Web. It can be tunneled over an encrypted SSL/TLS channel (HTTPS). DLP solutions employ mechanisms to access the clear text

 Copublished by the IEEE Computer and Reliability Societies 1540-7993/15/$31.00 © 2015 IEEE

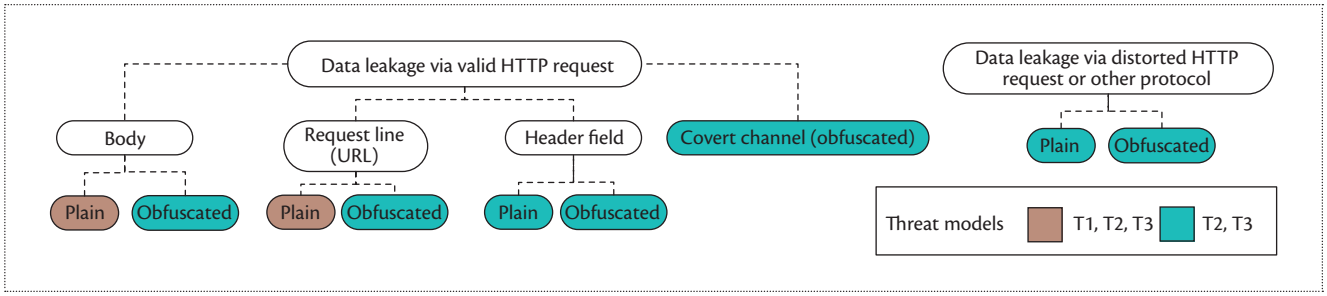

Figure 1. HTTP attack vectors. An attacker can follow the protocol semantics (left tree) or send arbitrary traffic over the HTTP port (right tree). In addition, malicious insiders and malware (threat models T2 and T3) might obfuscate data before inserting it into HTTP requests. T1, T2, and T3 refer to different threat models with different privileges and intentions in terms of data exfiltration.

of HTTPS traffic, which allows them to handle HTTPS traffic similarly to HTTP traffic. Thus, we treat HTTP and HTTPS as functionally equivalent and refer to both as HTTP.

HTTP can be used in many different ways to exfiltrate data. Figure 1 illustrates a systematic categorization of the possible data leakage vectors. At the top level, attackers might choose to follow the HTTP protocol syntax and semantics or send arbitrary traffic that violates the HTTP specification. Violating this specification might be a tempting strategy for leaking data as DLPs might not capture all possible types of distortion. However, it's generally easier to detect and block malicious traffic that violates the protocol semantics by simple protocol checking. In this article, we explain and analyze the challenges facing HTTP version 1.1. The upcoming HTTP version 2 faces similar issues, as it's semantically compatible.

HTTP/1.1 requests consist of the *request line* stating which resource is being requested from the server, a *message header* with additional information, and an optional *body*. Attackers following the HTTP protocol basically have four options to transmit data in HTTP requests: request body, request line, header fields, and covert channels.

Leakage in Request Body

This is the most straightforward case, as transmitting text input and files in the request body of POST requests is the recommended way to upload data to servers. According to the HTTP specification, uploaded data is either URL encoded or submitted as multipart/form data. A DLP solution can easily revert these encodings and access the original payload. Therefore, we refer to this subcategory as *plain* (as shown in Figure 1). Figure 2a shows an example submission of text input, and Figure 2b shows a file upload to a webpage. However, attackers might obfuscate the data they're trying to exfiltrate. They could achieve this by using other types of unforeseen encodings in the body or by using encryption. We combine encryption and unusual encodings in the *obfuscated* category in Figure 1.

```
POST /cgi-bin/u.cgi HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 ... Gecko/20100101 Firefox/20.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de-de,de;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://example.com/
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 45

secret=one%2C+two%2C+three
```

(a)

```
POST /cgi-bin/u.cgi HTTP/1.1
Host: example.com
...
Content-Type: multipart/form-data; boundary=...7963
Content-Length: 136

--...7963
Content-Disposition: form-data; name="secret"; filename="data.txt"
Content-Type: text/plain

one, two, three
--...7963--
```

(b)

```
GET /cgi-bin/u.cgi?secret=one%2C+two%2C+three HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 ... Gecko/20100101 Firefox/20.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de-de,de;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://example.com/
Connection: keep-alive
```

(c)

Figure 2. Example HTTP requests uploading secret=one, two, three to the server. Note that one%2C+two%2C+three is the URL-encoded representation of one, two, three. (a) POST request with URL-encoded data in the body. (b) POST request with multipart/form data file upload in the body. (c) GET request with URL-encoded data in the URL.

Leakage in Request Line

In HTTP/1.1, the requested URL is split into two parts. The Internet host part of the URL (example.com) is specified in the host header field, and the path of the resource on the server, including additional parameters,

is contained in the request line. Figure 2c shows an example GET request that uploads text input in the URL.

Leakage in Header Fields

In this slightly subtle case, attackers exploit a field of the request header to upload data. For example, data might be hidden in the Accept-Language field. Instead of defining the languages, attackers insert sensitive data. Attackers might include data as plain text or obfuscate leaked data through different encodings or encryption.

Leakage via Covert Channels

Covert channels are the most challenging strategy to defend against because leaked data isn't directly transmitted in the URL, header, or body. Instead, leaked data is hidden by a secret coding scheme that relates, for example, to the presence or order of header fields in transmitted requests, the timing of requests, or the selection of particular links when browsing.[6–9] Unless an organization is aware of such side channels, this type of leakage is extremely hard to prevent. In fact, we aren't aware of any DLP solution on the market that claims to be able to deal with this threat.

Data Leakage Threats

The ability to abuse different parts of the HTTP protocol for data exfiltration depends greatly on an attacker's access privileges and intentions. We take this into account by introducing three threat models (T1, T2, and T3) with different privileges and intentions in terms of data exfiltration. For each threat model, we show which parts of the HTTP protocol request messages could be exploited to leak data.

T1: Data Loss through Regular Web Browsing

In this threat model, sensitive data is intentionally or unintentionally leaked through regular Web browsing. We assume that data leaked this way isn't obfuscated before being transmitted over HTTP. As highlighted in Figure 1, the leakage vectors for T1 are limited to plain body and plain URL, as these are the two mechanisms expected to transmit user data to a server. For example, employees might accidentally leak sensitive information by uploading a file to their company's Facebook account. Such unintentional data leaks are very common: negligence and system glitches account for about half of all data breaches.[1]

T2: Malicious Data Exfiltration Using the Web Browser

In this threat model, we assume that attackers try to leak sensitive data using the browser. Attackers might exploit all browser features for exfiltration (such as JavaScript) but don't have the privileges to access the Internet with any other software. This is a typical scenario in security-aware organizations with managed workstations. Therefore, T2 represents the threat model of frustrated employees who try to leak confidential information.

Under T2, we have to assume that attackers can exploit all HTTP attack vectors. By creating a website with built-in JavaScript, it's trivial to obfuscate the data to be transferred by encoding it in the browser. With JavaScript, it's also possible to alter HTTP header fields and create modified or even invalid HTTP. This type of threat model might require programming skills to stage an attack, but the cost can be considered low because software exploits aren't necessary.

T3: Unconstrained Data Exfiltration

In this threat model, we assume that attackers or malware have gained root privileges on a workstation. Thus, attackers have unconstrained access to the workstation and could generate arbitrary traffic with a program of their choice. Data transfers might be obfuscated by means of encryption, side channels, or invalid HTTP. This is the highest threat level of the three. Because attackers or malware with root privileges could start and stop any processes on the workstation, we must consider the possibility that attackers might even disable a DLP agent running on the workstation.

DLP Solution Testing

There are two main types of DLP solutions: *host based* and *network based*. Host-based solutions must be installed on every workstation and include a combination of control mechanisms to disable or monitor network I/O, removable media, copy and paste, screen capturing, and printing as well as control which applications are executed or get network access.[10] Network-based DLP solutions analyze network streams on a gateway. We use the term *security control point* to refer to the locations of systems' control mechanisms.

When controlling dataflows at a security control point, the DLP solution might either block all dataflows to a destination (such as disabling copying to USB flash drives) or selectively identify sensitive information and block corresponding data transfers. Content-based DLPs identify sensitive information by searching the content of transferred information for patterns and keywords, often relying on predefined patterns for typical sensitive information—such as Social Security numbers, email addresses, or credit card numbers—and configurable keywords, such as "internal" or "classified." DLPs include parsers for a wide range of document formats—including Microsoft Excel, Word, and PowerPoint documents or PDF documents—and file containers (such as .zip, .gzip, and .7z) to extend their search to the contents of these documents and containers.

Testing Methodology

As is the case for many proprietary security tools, DLP vendors rarely provide documentation of their tools' internal implementations, much less the source code. This makes it difficult to determine whether a solution can protect against a particular attack. One possible approach is to reverse-engineer the executable. However, reverse engineering takes an exceptional amount of effort that would be annihilated by every software update released by the vendor. Thus, we developed a dedicated penetration-testing tool—called Data Exfiltration Tester (DXT)—that implements several data exfiltration methods over HTTP. Our black-box testing has an advantage over reverse engineering in that any system administrator can quickly detect flaws in a DLP solution's implementation or configuration by simply testing whether it prevents a particular kind of data leakage.

To analyze the implemented security controls, we took advantage of the fact that DLP solutions log or explicitly inform users when an action has been blocked. By correlating DLP alerts with actions conducted by DXT, we could gain insight into the kind of security controls used by a DLP solution. For example, if a DLP solution triggers an alert while our tool is reading a file from the file system, we can deduce that the file system I/O is being monitored. If the alert occurs only when a file is already loaded or when copied and pasted data is sent over the network, we can deduce that the network I/O is being monitored.

By combining multiple tests, we can deduce with high confidence which security controls are in place without reverse engineering. Thus, black-box testing provides an excellent tradeoff between analysis efficiency and confidence in the results.

To evaluate DLP solutions' capabilities, we installed each solution on a workstation and used DXT to exfiltrate test data from the workstation to the Internet over HTTP. DXT contains a component written in JavaScript (jsDXT) that obfuscates and uploads data directly in the Web browser as well as a Python-based stand-alone client (pyDXT) that obfuscates and uploads files independently from the Web browser—similar to how malware works. Both components implement methods to transmit plain and obfuscated data according to the attack vectors.

We used the same rules (R) to test all DLP solutions:

- R1—block documents containing any email addresses using a predefined pattern provided by the DLP,
- R2—block documents containing the keyword VH2uJKLi, and
- R3—block documents containing the keyword c/M!=&zu.

R1 tests one of the predefined patterns, whereas R2 and R3 test the ability to find keywords in data. We used two tests to evaluate whether URL encoding of text fields—the most prevalent parameter encoding (see Figure 2)—was properly handled. The keyword in R2 can be recognized even if URL-encoded values aren't decoded. To match R3, URL-encoded values must be properly decoded due to the special characters contained in the keyword. For each rule, we created matching documents of different types (.txt, .docx, .xlsx, and .pdf) as well as compressed versions of the documents (.gzip, .zip, and .7z). To evaluate a DLP solution according to threat models T1 and T2, we opened jsDXT in the Web browser and uploaded these documents to a Web server. In addition, we copied and pasted text to a form field and uploaded the text using jsDXT. For threat model T3, we submitted the documents using pyDXT.

Evaluation of DLP Solutions

We applied our testing methodology to three major vendors' host-based DLP solutions (referred to here as DLP A, B, and C). The vendors and solutions remain anonymous, as our goal isn't to pinpoint individual products but to showcase different DLP mechanisms and discuss their effectiveness. *SC Magazine* and *Network World* evaluated some of the solutions and gave them high ratings. However, those evaluations reflect case studies that lack an exhaustive attack vector classification and a clearly defined threat model. The vendors are also listed in market reports by Gartner and the Radicati Group.[4,5]

Tables 1 and 2 show details of the applied attack vectors with exfiltration of copied and pasted text via a Web form (Table 1) and with exfiltration of different file types via a Web form/script (Table 2). Each row represents an HTTP attack vector with the vertical labels (T1 to T3) showing the corresponding threat models. The columns show the tested DLPs and uploaded file types. A green circle means data loss was prevented (all three rules were effective). A white circle means data loss was partially prevented (R1 and R2 were effective but R3 failed). A red square means there was complete data loss (all rules failed).

In Table 1, attack vectors 1 through 4 (threat model T1) submitted the plain data. In attack vectors 5 through 9 (threat model T2), the HTTP requests were slightly obfuscated by placing data at unexpected positions in the HTTP request (attack vectors 5 and 8) or by base64 encoding the data prior to submission. In Table 2, attack vectors 10 through 12 (threat model T3) submitted the data using pyDXT, first plain, then base64 encoded, and finally plain but as distorted HTTP without any request line or header. Most benign websites transfer text input as shown in attack vector 1 and file uploads as shown in attack vector 2. Thus, these two attack vectors are most relevant with respect to threat model T1.

Table 1. Exfiltration of copied and pasted text via Web form for three data loss prevention (DLP) solutions.

Threat model	Attack vector no.	Attack vector type	DLP A	DLP B	DLP C
T1	1	Body plain: URL encoded in POST body	○	■	○
T1	2	Body plain: multipart/form data in POST body	○	■	○
T1	3	Body plain: raw data in POST body	○	■	○
T1	4	URL plain: URL encoded in request line of GET request	○	■	○
T2	5	URL plain: URL encoded in request line of POST request	○	■	○
T2	6	Body obfuscated: base64 and URL encoded in POST body	○	■	○
T2	7	Body obfuscated: base64 encoded in multipart/form data POST body	○	■	○
T2	8	Header field plain: in additional header field of GET request	○	■	○
T2	9	Header field obfuscated: base64 encoded in additional header field of GET request	○	■	○

○: data loss prevented (all rules effective)
■: complete data loss (all rules failed)

Results

Our results show that DLP A was able to successfully block all data leakage where adversaries tried to copy and paste data to a text field in the Web browser. HTTP requests with files in multipart body containers were blocked when submitted through the Web browser or through pyDXT. However, 7z-compressed files didn't seem to be supported. Obfuscated file uploads weren't blocked at all. This suggests that DLP A controls the clipboard and network I/O on the workstation. However, as experiment 3 showed, the network analyzer didn't support files that were transmitted without encapsulation in multipart containers.

DLP B did not prevent any data leakage through text form fields. File leakage using attack vectors 2, 3, and 7 was prevented as the Web browser was denied access to files matching the rules. However, pyDXT can read and upload all files. These results strongly suggest that DLP B controls only the Web browser's file system I/O. The implementation seems to be flawed, which leads to failure in enforcing R3 with .docx and .xlsx files. Controlling files when they're read by an application instead of controlling the HTTP requests sent by the application is advantageous in that obfuscation in the application has no effect on the DLP solution. Thus, DLP B can prevent plain and obfuscated file uploads through the Web browser. However, pyDXT isn't monitored by DLP B, and accidental copying and pasting of sensitive information can't be prevented because the clipboard and the network traffic don't seem to be monitored.

DLP C blocked all data leakage attempts in which adversaries tried to copy and paste data to a text field in the Web browser. Data leakage through the most commonly used upload mechanisms (attack vectors 1 and 2) was also blocked. Raw file uploads in POST requests (attack vector 3) were partially blocked. The unobfuscated data submitted by pyDXT was successfully blocked. Obfuscated file uploads weren't blocked at all. These results suggest that DLP C monitors the clipboard and network I/O on the workstation.

To summarize, we found that all three DLP solutions can prevent accidental file uploads over HTTP if attack vector 2 is used. However, none of the solutions can prevent data leakage under threat models T2 and T3. For example, to bypass DLP A and DLP C, T2 adversaries can simply obfuscate a document in JavaScript; to bypass DLP B, T2 adversaries can copy and paste the contents to a text input field in the Web browser. Furthermore, we found flaws in the implementation of security control points that resulted in problems enforcing a rule, even when a control point was in place.

Security Control Points

Here we look at the flawed combinations of security control points implemented by the evaluated DLPs and present security control combinations that would more efficiently prevent leakage.

Implemented Security Control Points

By correlating DLP alerts with actions conducted by our black-box testing tool, we found three security control points (shown in Figure 3):

Table 2. Exfiltration of different file types via Web form/script.

			DLP A							DLP B							DLP C						
Threat model	Attack vector no.	Attack vector type	.txt	.docx	.xlsx	.pdf	.txt zip	.txt gz	.txt 7z	.txt	.docx	.xlsx	.pdf	.txt zip	.txt gz	.txt 7z	.txt	.docx	.xlsx	.pdf	.txt zip	.txt gz	.txt 7z
T1	2	Body plain: multipart/ form data in POST body	●	●	●	●	●	●	■	●	○	○	●	●	●	●	●	●	●	●	●	●	●
T1	3	Body plain: raw data in POST body	■	■	■	■	■	■	■	●	○	○	●	●	●	●	●	○	●	■	●	■	●
T2	7	Body obfuscated: base64 encoded in multipart/form data POST body	■	■	■	■	■	■	■	●	○	○	●	●	●	●	■	■	■	■	■	■	■
T3	10	Unknown application body plain: multipart/ form data in POST body	●	●	●	●	●	●	■	■	■	■	■	■	■	■	●	●	●	●	●	●	●
T3	11	Unknown application body obfuscated: base64 encoded in multipart/form data in POST body	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
T3	12	Unknown application distorted HTTP: raw data without a request line or header	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■

●: data loss prevented (all rules effective)
○: data loss partially prevented (rules 1 and 2 effective)
■: complete data loss (all rules failed)

- *File system I/O on the workstation (FILE_I/O)*—control of files that are read from disk or network storage.
- *Copy and paste on the workstation (CLIPBOARD)*—control of data that is copied between applications via the clipboard.
- *Network I/O on the workstation (NET_HOST)*—control of HTTP and HTTPS network traffic. By installing a hook in the network API, a DLP solution can inspect and block requests prior to leaving the workstation.

For comparison, we also considered a security control point that corresponds to a network-based product:

- *Network I/O on a network gateway (NET_GATEWAY)*—control of HTTP and HTTPS network traffic. Network-based DLP solutions employ an HTTP proxy server to monitor data streams. The proxy server intercepts SSL/TLS-encrypted data streams to provide clear-text access to HTTPS requests. Detected data leaks can be avoided by terminating the corresponding network stream.

Table 3 summarizes which of these security control points appear to be implemented by the evaluated DLP solutions. The security control point labels correspond to Figure 3.

Mandatory Security Control Points

Here we relate the three threat models to a minimum set of security control points required to prevent data leakage over HTTP. In other words, assuming that a DLP policy for email addresses has been defined, we show for each threat model which security control points must be used to prevent leaking a document containing email addresses. Table 4 shows mandatory security controls to protect against a particular threat model as

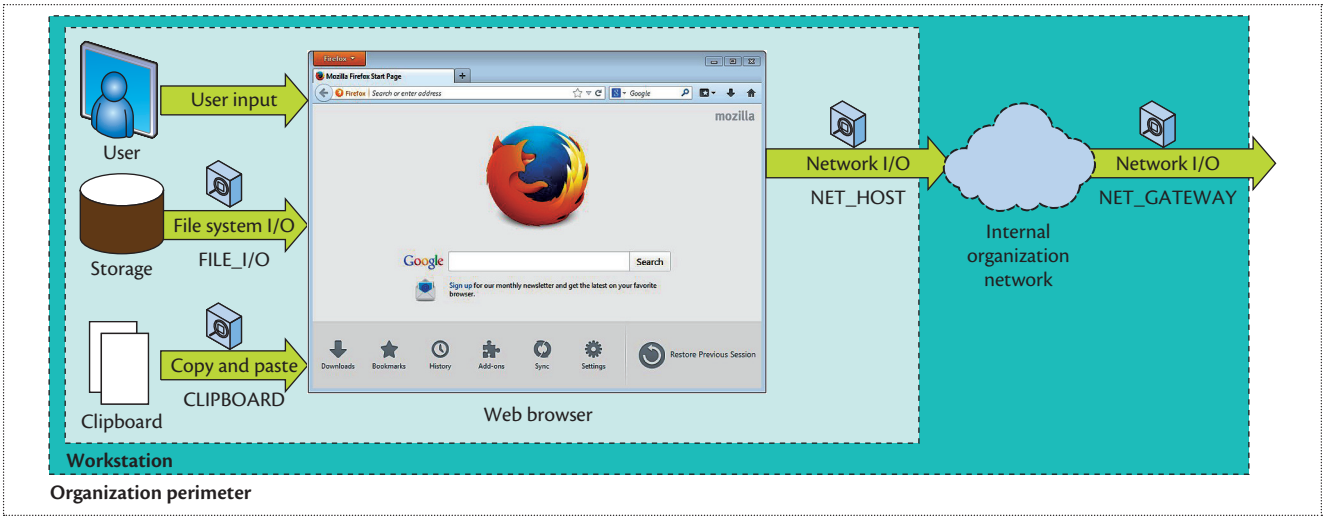

Figure 3. Security control points for enforcement of DLP rules for Web browsing. Data can be provided to the Web browser as user input via the keyboard, the file system, or the clipboard. Security control points FILE_I/O and CLIPBOARD monitor the file system I/O and the clipboard, respectively. NET_HOST controls network traffic by monitoring the workstation's network API, and NET_GATEWAY monitors the organization's network traffic.

Table 3. Security control points.

Solution	Implemented security controls (derived from black-box testing and product documentation)
DLP A	CLIPBOARD and NET_HOST
DLP B	FILE_I/O (only for supported applications)
DLP C	CLIPBOARD and NET_HOST

Table 4. Mandatory security controls.

Threat model	Prevention	Detection and forensics
T1: regular Web uploads	FILE_I/O and CLIPBOARD or NET_HOST or NET_GATEWAY	NET_HOST or NET_GATEWAY
T2: malicious exfiltration using Web browser	FILE_I/O and CLIPBOARD	NET_HOST or NET_GATEWAY
T3: malware	There is no sufficient combination of the identified security control points	NET_GATEWAY

well as security control points for network-based leakage detection and forensics.

For threat model T1, the Web browser either doesn't transform the data before uploading to a Web server (multipart/form data) or it transforms the data as specified by well-defined standards (URL encoded) so that network-based monitoring solutions can reverse the data transformations. As a result, employing a network-based solution, such as NET_HOST or NET_GATEWAY, is sufficient. Alternatively, a DLP can rely on a combination of FILE_I/O and CLIPBOARD to control file uploads and copied and pasted data.

For threat model T2, data can be obfuscated in the Web browser before being sent to a Web server. Because obfuscations can be arbitrary, it generally isn't feasible to know which obfuscations have been applied or to reconstruct the original data from network traffic. For example, in attack vector 7 in Table 2, we applied base64 encoding to files as a very simple means of obfuscation. Yet, this was enough to allow our HTTP requests to pass through the network I/O security control points of DLPs A and C. Attackers could use much more sophisticated obfuscation or even encrypt (http://code.google.com/p/crypto-js) and obfuscate data in JavaScript. Thus, internal attackers could use a website with JavaScript to evade purely network-based DLP solutions

(such as NET_HOST or NET_GATEWAY). For effective data leakage prevention, a DLP solution must employ security control points that can inspect data before obfuscation, such as FILE_I/O and CLIPBOARD, which control data before it's processed by the Web browser. Note, however, that even this combination of security control points can't prevent data leakage if another application has already obfuscated the data before it's copied onto the clipboard or read from the file system.

Malicious software in threat model T3 can run with root privileges on a compromised workstation. The malware might tamper with or disable any DLP solution running on a workstation, rendering the security control points FILE_I/O, CLIPBOARD, and NET_HOST ineffective. However, malware on the workstation can't disable a security control point in the network. Therefore, NET_GATEWAY can prevent unobfuscated data leakage, but as soon as the malware obfuscates data transfers, NET_GATEWAY is also rendered ineffective. As a result, once malware with root privileges is running on a workstation, it's generally not feasible to enforce a DLP policy. Even though NET_GATEWAY can't prevent obfuscated data leakage, it can still provide useful connection information for detection and forensics.

We found that DLPs A, B, and C all implement security control points that can prevent accidental file uploads over HTTP. However, none of the solutions employ the correct combination of the mandatory security control points to prevent basic data leakage caused by internal attackers or malware. Thus, these solutions can't protect against T2 or T3. By following our recommendations regarding the correct combination of mandatory security controls, these DLP solutions would be significantly improved.

Our study suggests that this deficiency is related to the lack of fundamental security controls at the required vantage points. Only one of the evaluated DLP solutions transparently reports this severe constraint. One vendor even explicitly states that its product addresses data leakage caused by internal attackers or malware, which we found to be false. We invite organizations to test their DLP solutions to make sure they meet the promises found in vendors' product descriptions. ■

Acknowledgments

This work was partially supported by the Zurich Information Security Center at ETH Zurich and represents only the views of the authors.

References

1. "2015 Cost of Data Breach Study: Global Analysis," Ponemon Inst., May 2015; www-03.ibm.com/security/data-breach.
2. "Top Banking Botnets of 2013," Dell SecureWorks, 3 Mar. 2014; www.secureworks.com/cyber-threat-intelligence/threats/top-banking-botnets-of-2013/.
3. "Defense in Depth: A Practical Strategy for Achieving Information Assurance in Today's Highly Networked Environments," Nat'l Security Agency; www.nsa.gov/ia/_files/support/defenseindepth.pdf.
4. "Magic Quadrant for Endpoint Protection Platforms," Gartner, 2 Jan. 2013; www.gartner.com/doc/2292216/magic-quadrant-endpoint-protection-platforms.
5. "The Radicati Group Releases 'Content-Aware Data Loss Prevention Market, 2013–2017,'" Radicati Group, 11 Mar. 2013; www.radicati.com/?p=9377.
6. N. Feamster et al., "Infranet: Circumventing Web Censorship and Surveillance," *Proc. 11th USENIX Security Symp.*, 2002, pp. 247–262.
7. S. Cabuk, C.E. Brodley, and C. Shields, "IP Covert Timing Channels: Design and Detection," *Proc. 11th ACM Conf. Computer and Comm. Security* (CCS 04), 2004, pp. 178–187.
8. N. Schear et al., "Glavlit: Preventing Exfiltration at Wire Speed," *Proc. 5th Workshop Hot Topics in Networks* (HotNets 06), 2006, pp. 133–138.
9. K. Borders and A. Prakash, "Quantifying Information Leaks in Outbound Web Traffic," *Proc. 30th IEEE Symp. Security and Privacy*, 2009, pp. 129–140.
10. A. Shabtai, Y. Elovici, and L. Rokach, *A Survey of Data Leakage Detection and Prevention Solutions*, Springer, 2012.

David Gugelmann is a doctoral student at ETH Zurich. His research interests include network forensics and privacy measurements. Gugelmann received an MS in electrical engineering and information technology from ETH Zurich. Contact him at gugelmann@tik.ee.ethz.ch.

Pascal Studerus is a software engineer at AdNovum Informatik AG. Studerus received an MS in computer science focusing on information security from ETH Zurich. Contact him at pascalstuderus@gmail.com.

Vincent Lenders is a research program manager at armasuisse, where he is responsible for the Cyberspace and Information research program. Lenders received a PhD from ETH Zurich and was a postdoctoral researcher at Princeton University. He serves as the industrial director of the Zurich Information Security Center at ETH Zurich. Contact him at vincent.lenders@armasuisse.ch.

Bernhard Ager is a senior researcher and lecturer at ETH Zurich. His research is driven by his desire to understand large and complex systems such as the Internet. Ager received a PhD from TU Berlin and was a researcher at Deutsche Telekom Laboratories (T-Labs). Contact him at bager@tik.ee.ethz.ch.