Application-screen Masking: A Hybrid Approach

Abigail Goldsteen, Ksenya Kveler, Tamar Domany, Igor Gokhman, Boris Rozenberg, Ariel Farkash
Information Privacy and Security, IBM Research – Haifa
{abigailt,ksenya,tamar,igorgok,borisr,arielf}@il.ibm.com

***Abstract*—Large organizations often face difficult trade-offs in balancing the need to share information with the need to safeguard the privacy and security of sensitive data. A prominent technique for dealing with this trade-off is on-the-fly screen-masking of sensitive data in applications. In this paper we present a unique hybrid approach to screen-masking by combining the advantages of the context available at the presentation layer with the flexibility and low overhead of masking at the network layer. Our solution enables the identification of sensitive information in the visual context of the application screen, then automatically generates the masking rules to be enforced at runtime on the network traffic. This approach is more powerful and user-friendly than the regular expression based mechanism typically employed by traditional network-based solutions. We show that our approach supports the creation of highly expressive masking rules, while keeping the rule-authoring process easy and intuitive, thus resulting in a system that is both easy to use and effective.**

I. Introduction

Large organizations often face difficult trade-offs in balancing the need to share information with the need to safeguard the privacy and security of sensitive data. They must share data, both internally and externally, to remain competitive, yet regulations and client expectations often restrict the exposure of sensitive information.

Organizational data is often accessed using software applications. An application can be accessed by different users for a variety of purposes over its life-span. For example, new users may be introduced when outsourcing business processes, or regulations regarding what type of information can be exposed to whom may change. As a result, the privacy and security needs of the application evolve and require changes to existing access control mechanisms.

As a concrete example, assume an insurance company based in Germany needs to outsource its claims processing center to India for financial reasons. The European Data Protection Directive 95/46/EC [1] imposes restrictions on the transfer of personal data from European countries to third countries. Therefore, to allow call-center agents located in India to access the company's web application, all sensitive personal information must be removed from application screens.

The hiding of sensitive information can be implemented in various ways. Some claim that the best approach is to re-engineer the application to ensure that no sensitive data is transmitted at all, using techniques such as Program Analysis [2] or Aspect Oriented Programming [3]. These techniques are used to either verify that an application does not expose any sensitive data or to rewrite an application to support new privacy requirements. These methods, however, can be costly, time-consuming and error-prone. Moreover, the original application designers are not always available, and locating someone with the appropriate skills to implement this strategy is not always feasible.

An alternative approach is to mask the sensitive data that is being displayed by the application at the data layer. Databases can be copied, and sanitization techniques applied to their contents to either remove or transform parts of the data to match new requirements [4]. However, when sharing applications with third-party users, an organization could want to restrict certain users from viewing certain pieces of information while enabling other users to see them. Thus, the organization would need to create several copies of the database for each involved entity, which would be onerous to create and maintain, if at all feasible.

Masking at the application-screen level can be used to hide sensitive information without interfering with the application that generated those screens. This is done by introducing an additional layer between the application and the end-user that operates on the application screens. Screen masking can be used to mask not only sensitive data but also sensitive actions, such as preventing clicking on a web link and thereby reaching a sensitive page. Thus, this feature can provide an additional layer of security and control over what users can see or do. Because only the display data is changed, another benefit of using this method is that the application itself still contains the original data and can interact with it, thus enabling scenarios in which it is required to reveal the masked data due to some urgent business or ethical need, known as "breaking the glass" [5]. This is sometines the case in medical scenarios when it is imperative to see a patients record in order to save his life.

There are several ways in which a screen masking policy can be defined. We divide the masking rules into two basic types: **Content-based rules** that take into consideration only the content of the text and can be defined either by regular expressions or more advanced text analytics tools; and **Context-based rules**, that are based on the visual structure of the screen, i.e., the presentation layer. This means that a rule author navigating the application can, figuratively speaking, tap a finger on the screen and say: "I need this column masked" or "I want to mask the field next to this label". Context-based rules can be based on UI constructs, such as labeled fields, table columns, drop-down boxes, etc., or defined by a relationship between two entities on the screen or by their absolute locations.

An example of a context-based rule would be to mask all labeled fields in which the label is "Email Address", as depicted in Figure 1. A content-based rule may simply contain a regular expression depicting email adresses. In this case, all emails will be masked, without the need of a specific label.

Content-based rules are more straightforward to define and

Fig. 1. Context-based labeled field rule

enforce. For context-based rules, the task is more complicated. The gap between the concepts with which the administrator defines the rules, i.e., what is seen on the screen, and the code that executes at runtime, is much wider. Context-based rules must somehow match between entities at the presentation layer (e.g., objects on the screen) and instructions that are executed at runtime (such as an exact coordinate or XPath [6]). For example, a rule author who wants to mask a table column with the header "Phone", as seen in Figure 3, needs to translate this into a formal instruction set that will implement the runtime masking.

Context-based rules give more flexibility than content-based rules. An example would be masking only home phone numbers and not work phone numbers. The tradeoff in creating this kind of rule is the need to formulate several rules to cover all instances in the application of home phone numbers (for example, if they appear both in a form and in a table), whereas a content-based rule can cover all phone numbers in one rule.

Several methods exist for implementing masking at the application-screen level. One method uses network-traffic inspection, or "protocol-sniffing" [7], to intercept data as it flows through the network toward the client machines and then analyzes and alters it. Current protocol-sniffing solutions are efficient, but they offer only simple content-based masking rules. A different approach is to focus on the presentation layer, using Optical Character Recognition (OCR) [8] (for more details see Section II). In this method, the screen is captured as an image, then analyzed and modified before being displayed on the end-user's screen. While this method provides powerful capabilities for context-based rule definition, it suffers from difficulties in handling complex screens and severe performance issues.

In this paper, we present a novel method for performing context-based screen-masking, that can conceal sensitive data from specific user roles, without requiring any changes to the existing application or data stores, and without impacting the application's functionality or the end-user's experience. We address applications that are delivered from a server to any client software, with particular focus on web applications.

We tackle cases in which it is necessary to conceal sensitive information on screens in a way that is transparent to the end-users operating them, while striving to minimize the performance impact. We use the protocol-sniffing approach to perform the masking, which does not require any changes to the existing application or data stores, nor does it require any installation on the end-user's machine.

Our main contribution is a hybrid approach that combines enforcement at the network level with powerful context capabilities resulting from defining the masking rules at the presentation level. We provide an intuitive, visual rule-authoring process, that does not require great technical expertise, making it easy to create and modify masking rules. Our network-based implementation has negligible impact on runtime performance. This results in a system that is both easy to use and effective. Combining visual context capabilites with masking at the network level is a novel approach which presented several technical difficulties that are discussed in later sections.

This paper is organized as follows: Section II describes related work, Section III describes details of our approach and Section IV discusses the advantages and drawbacks of our solution compared to the alternatives. Section V includes some performance results. We summarize our implementation and suggest directions for future work in Section VI.

II. Related work

The subject of privacy, security, and integrity of web applications has received much research attention. A large volume of work deals with identifying vulnerabilities [9][10] and faulty input sanitization procedures [11] using various code analysis techniques. Side-channel weakness is analyzed in [12] to demonstrate that sensitive information is being leaked from web traffic despite encryption.

Techniques for proactive security and privacy integration into applications have also been suggested. The Servlet Information Flow (SIF) framework [13] can be used for building high-assurance web applications and using language-based information-flow control to enforce the appropriate release of confidential information to clients. In cases in which it is not possible to proactively integrate security and privacy into the design of the system because of economic, practical, or historical reasons, retroactive program analysis techniques and tools can assist in retrofitting appropriate mechanisms into legacy code [14] as the need arises.

A number of research efforts and commercial products deal with sensitive-data leakage prevention by modifying data stores. These techniques and tools usually hide sensitive data in databases by systematically removing or transforming their contents, in a way that keeps data realistic yet de-identified [15]. Sophisticated data masking algorithms are employed [16] to ensure that dataset-level properties and statistics remain approximately the same, allowing for research and data mining. Commercial products such as IBM Optim [17], Oracle Data Masking [18], Camouflage [19], and Voltage SecureData Masking [20] offer data-masking capabilities while preserving data usefulness and referential integrity.

Unfortunately, methods such as proactively designing applications with privacy in mind, re-engineering legacy applications, or masking data stores, are costly and not always feasible. Existing applications, especially legacy programs, are rarely re-written, and maintaining separate copies of the application database for different user roles can be very difficult. Changes to the underlying database may also impact the application's functionality. In such cases, on-the-fly application-screen masking is a potential solution.

Verdasys Digital Guardian Application Logging and Masking Module [21] provides screen-masking support on Windows platforms. However the Verdasys technology depends on a software component which must be be installed on every client machine where screen-masking capabilities are required. In addition, client-side solutions are considered less safe as the sensitive information arrives at the client machine and is masked there.

Another screen-masking method uses OCR [8] as the core technology to capture, analyze, and mask application screens. Screens are intercepted at the point where the screen image is rendered, and then rerouted to discover and mask the sensitive texts before displaying them to the end user. This method is independent of the protocol and platform, but it has many challenges in recognizing entities on the screen due to overlapping, scrolling, and other complex screen structures. This technique also requires that operations like copy&paste or print-screen be prevented to avoid revealing the sensitive information. However, the main drawback of this method is the performance impact.

Network-traffic inspection is a widely used method for application-screen masking and other purposes. The Intellinx Enterprise Fraud Detection and Prevention solution [22] employs network-traffic sniffing to record end-user interactions for auditing and fraud detection. This solution allows the masking of recorded screens to prevent an auditor from seeing sensitive information. However, to the best of our knowledge, no on-the-fly masking is performed while end-users are working with application screens. IBM Infosphere Guardium [23] employs network-sniffing techniques for real-time database security, monitoring, and auditing. Check Point DLP Software Blade™ [24] inspects data transmitted over networks in order to detect and avoid sensitive information loss. However this application does not seem to allow masking of the data in-motion. Sensitive data can be identified by their similarity to commonly-used templates, which is a primitive form of context-based rules, but with a much smaller scope and flexibility than our approach. They also use a scripting language for tailoring custom cases, but it suffers from severe usability issues.

Privacy Infrastructure Appliance (PIA) [25], inspects communications to anonymize sensitive data sent from service takers to service providers, without changing the application. However, this appliance deals with the use of a third-party application supplied by a service provider, where the sensitive data can be seen and manipulated by the application users, but cannot be stored in the application database. The system also requires sensitive data tagging, so that it can be identified and replaced with masking tokens at runtime. Riverbed® Stingray™ Traffic Manager [26] provides application-screen masking by network-traffic inspection. It employs content-based string-matching techniques, such as regular expressions, to define the targets to mask. The approach we present in this paper enables the definition of comprehensive context-based masking rules, which take into consideration UI constructs without any explicit tagging.

III. Our approach

In this section, we describe our solution for application-screen masking with context-based rules. Our approach is based on intercepting network messages sent between a server and a client and altering them according to rules.

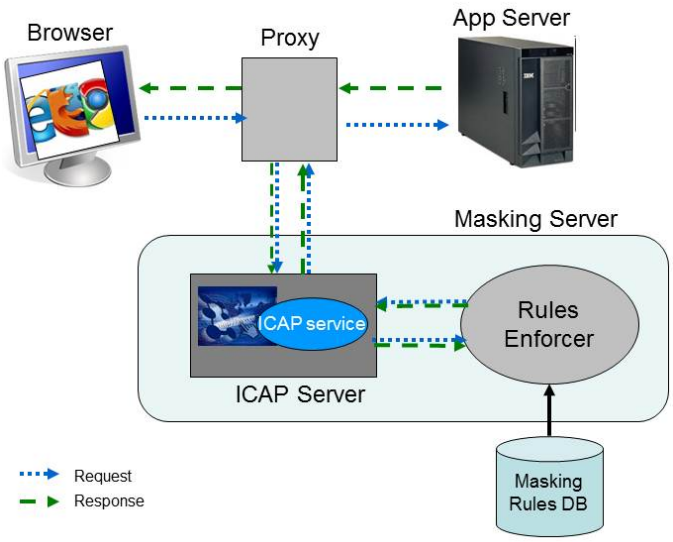

Fig. 2. High level architecture

A. Solution design

The core component of the system is a “sniffer”, which intercepts all requests sent from the client to the server and all response messages sent from the server to the client. For each response that carries information for display, a rule set is traversed to check if any masking rule should be applied. If such a rule is found, the response is altered according to the rule before it is sent to the client. Note that the sensitive information is completely removed from the message and does not reach the client machine. The client that receives the messages renders the screen to the display.

The client requests are also intercepted to check if they include information that was previously masked. If so, the request is reconstructed with the correct data, meaning the masked data in the request is replaced with the original data, which was saved in the “sniffer”, before being sent to the server. This way the application gets the correct data in the request, and we do not “break” the application.

In the first phase of the work we focus on web applications, thus restricting ourselves to dealing with the HTTP and HTTPS protocols out of the many available application networking protocols. Figure 2 illustrates a sample architecture for web applications. The network is configured to send all communications between the application server and the client browsers through a proxy server. The proxy then passes the messages to an ICAP [27] server. An ICAP service parses the message headers and passes the relevant information to the enforcer component. The enforcer uses the message headers to search for relevant rules in the rule set. If it finds one or more rules that should be applied to the message, it parses the payload and enforces those rules.

Since all sensitive information is removed from the message, it does not reach the browser, and thus cannot be revealed by the end user, even when performing ’view source’. In

addition, both the masking server and the proxy itself are placed within the enterprise's internal network or firewall, thus preventing any sensitive information from leaving the premises.

Our implementation uses several Open Source components. For the HTTP proxy capabilities, we use the Squid proxy [28], for the ICAP implementation, we chose c-icap [29] and added an ICAP service that transfers the messages to the rule enforcer.

HTTP message payloads may come in many different formats (e.g., HyperText Markup Language (HTML), Extensible Markup Language (XML), JavaScript, JavaScript Object Notation (JSON), plain text, etc.). After observing a representative sample of applications, we found that the most frequent data formats are HTML and XML. In newer applications, JSON has become prevalent. Consequently, we integrated parsers for HTML, XML and JSON, using the the Libxml2 [30] and Jansson [31] implementations respectively.

B. Rule language and enforcement

Placing our enforcement component on the network allows us to access and alter almost any piece of information that appears on the screen. When creating a masking rule, the rule author may choose to apply certain filters that affect which messages the rule will be applied to. Such filters may include the server or client IP addresses, a certain user or group of users, and a URL pattern. At runtime, the server and client IPs as well as the request URL are provided as part of the HTTP protocol. To identify the current user, the system recognizes the application login process and extracts the username from the corresponding message. That username is then bound to the current session until a logout is performed. All this information is taken into account when deciding whether to apply a rule.

Once the system has decided that a rule should be applied to a message, the information to mask must be identified within the message. Our masking system supports both content-based and context-based masking rules. This enables the rule author to select the more suitable type of rule according to his masking needs. For example, if all email addresses in the application need to be masked, using a content-based rule is best. On the other hand, if only a few email addresses should be masked (and others should not), a context-based masking rule is more suitable. Context-based masking is also appropriate for masking texts that do not have a pre-defined format, such as names. The focus of this paper is the context-based part of our rule language, since content-based rules are relatively straightforward to create and enforce.

A masking rule must also specify what type of masking to perform. There are numerous possibilites, ranging from simply removing the values, changing the visual representation (such as modifying the background color in addition to removing the text value), replacing the original value with a different fictitious one, and many more.

To achieve this level of flexibility, in addition to the major requirement of minimizing the impact on performance, we describe our rules in JavaScript and use the SpiderMonkey interpreter [32] to execute them. Each rule contains a JavaScript script describing the changes to be performed on a given

Actions Selected: 0

	Account Name	City	Billing Country	Phone
	5D Investments	Ohio	USA	
	5D Investments	St. Petersburg	USA	
	A.D. Importing Company Inc	Denver	USA	
	A.D. Importing Company Inc	Denver	USA	
	AB Drivers Limited	Alabama	USA	
	Air Safety Inc	San Mateo	USA	
	Airline Maintenance Co	San Jose	USA	
	Anytime Air Support Inc	Santa Fe	USA	
	AtoZ Co Ltd	Alabama	USA	
	Avery Software Co	Santa Fe	USA	
	B.H. Edwards Inc	Cupertino	USA	
	Davenport Investing	Kansas City	USA	

Fig. 3. Column masked by enforcing the script from Listing 1

message at runtime. This expressive scripting language enables specifying any type of context-based rule, including any screen construct and any type of relationship between elements on the screen, regardless of the data format. Listing 1 shows a script that can change a message where the payload carrying the data is in HTML. The result of executing it can be seen in Figure 3.

The fact that our solution resides on the network, can inspect all passing messages and employ scripts on them gives us fine-grain control over the masked elements and enables us to mask exactly what is needed. The limitation of such an approach is that we cannot mask information that does not flow over the network, i.e., that is generated on the client-side. An example of such information is an average that is calculated in the browser using Javascript.

```
var elements =
html.xpath("/html/body[1]/div[2]/div[1]/table[1]/
    tbody[1]/tr[1]/td[1]/div[2]/form[3]/table[1]/
    tbody[1]/tr/td[7]/text()");
for (n in elements) {html.mask(elements[n]);}
```

Listing 1. Masking script to cover a table column arriving in an HTML message

C. Visual rule-authoring

A usable masking system should allow the rule author to easily create a new rule or modify an existing one. However, it is generally the case that the more powerful a language is, the more complicated it is to generate rules.

Generating context-based masking rules can be significantly more complex than rules handled by traditional protocol-sniffing mechanisms, as they relate to how the information is displayed on the screen and not just to the content of the texts. Part of the problem stems from the fact that information flowing through the network is not easily mapped to a displayed element. A simple table appearing on the screen can arrive as several messages, possibly in different formats, each carrying a chunk of information, which is ultimately translated to a single table on the client side. For example, a table in a web application may arrive in three separate messages: one HTML message describing the column headers, fonts, and colors, the second message containing the actual data in JSON format, and a third message containing a script that generates totals and summaries for the table.

Defining these rules manually would result in a significant loss in usability, due to the expertise required for correlating the presentation layer with the underlying network traffic.

For example, to define a rule for masking a table column on a web application page, rule authors would first need to see how that page is presented by the browser. They would then check the page source or intercept network traffic messages to determine whether the table content is in HTML, or is built on demand by Asynchronous JavaScript and XML (AJAX) requests. The authors may be required to understand the association between Document Object Model (DOM) elements and the target column. They may also need to analyze the network message payload to discover exactly where the information to mask is located. After the masking target is isolated, the author still needs to create a masking script, validate its syntactic correctness, and confirm that the masking is performed correctly on the displayed page.

The rule author thus requires expertise in several disparate technologies and tools, and is involved in a lengthy error-prone process. To overcome this difficulty, we propose a hybrid approach that enables creating rules in the "language" of the presentation even though they are enforced in the "language" of the protocol, using a visual rule-authoring tool to aid in this complicated task.

Our rule-authoring tool is a visual editor that enables the creation of powerful context-based rules in an intuitive and user-friendly manner, while navigating the target application screens and selecting areas to mask. A panel attached to the application enables the selection of a context on the presentation layer by pointing the mouse and indicating that this area (e.g., table column) is to be masked. The selection is then transformed into a machine-readable masking rule to be run during enforcement.

Figure 4 demonstrates our implementation of the visual rule-authoring tool for web applications. To avoid installation on authors' machines, we implemented a web-based tool, allowing authors to define rules using a web browser. In rule-authoring mode, authors can navigate the target application in a natural manner, emulating end-users' normal operation. When an area to be masked is selected (such as the table column selection in Figure 4), the tool performs a combined analysis of both presentation and network traffic data to automatically create JavaScript-based rules for masking the selection. By performing contextual analysis of the page structure, the tool is able to provide visual hints to the administrator, such as automatically expanding the selection to the whole table column when hovering over one of the table cells or when a certain cell is selected. After the selection is made and the rules are generated, the tool can show an in-place preview of the resulting masking, providing immediate feedback for rule validation.

Defining the masking rule in the visual context of the presentation layer allows an intuitive and humanly-manageable way to author masking rules. Transforming these rules to a machine-readable format to be applied at the network protocol level prevents common maladies of existing presentation-based methods, e.g., difficulty in handling complex screens and performance issues. Extensive user studies with real administrators are planned to test the usability of our rule authoring mechanism.

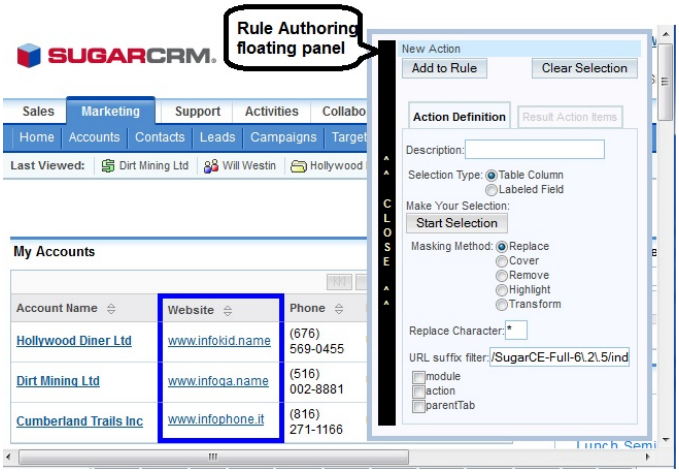

Fig. 4. Visual rule-authoring panel on top of a web application page with table column is selected

The main technical challenges in implementing the visual rule-authoring tool were:

- Automatically translating the visually described policies into machine-readable instructions
- Overcoming the security barriers so that the tool can interact with and inspect the target application

These issues are addressed below.

1. Automatic rule generation

The first task in automatic rule generation is determining the origin of each visual element that appears on the screen. This involves identifying the event that caused the display, and the source of the data content that is displayed

Each modification of an element on the screen can originate either from an incoming HTTP message (network-originated modification) or from some other activity, such as JavaScript code (locally-originated modification). It is not always straightforward to deduce which event caused an element on the screen to be created or modified. For example, JavaScript code activated on a timer event or as a result of user input could occur simultaneously with the arrival of an HTTP message.

It is also complicated to identify the data source of each screen element, i.e., to determine which message supplied the data, as well as detect the data location within the message. The source of the data may be any HTTP message received from the web server as a result of a GET/POST request from the browser, an AJAX request initiated by the web application itself, or even a value computed locally by the browser.

The method we will describe shortly enables:

- Automatic differentiation between network-originated and locally-originated modifications.
- Automatic identification of the data source (specific message) for each element presented on the screen.
- Automatic detection of the exact location of the presented data in the corresponding message.

Returning to the example in Figure 3, our method can automatically identify that the data in the “Phone” column comes from the specified HTTP message and create the masking script presented in Listing 1.

To acheive this, we monitor web page modifications in terms of DOM tree changes and capture only those changes on the web page that were initiated by HTTP messages (network-originated changes), while filtering out all other changes (locally-originated changes). The modified DOM elements that pass this filter are then mapped to the pieces of data within the HTTP messages that caused the change in these elements.

The following algorithm produces a map between visual elements presented in the browser and the HTTP messages and locations within the messages that contain the elements' data.

1) Capture the original HTML message that built the page and create a temporal DOM tree from its contents (not including script tags).
2) For each element in the tree, save the URL of the message it originated from and the location of the element within the message (e.g., XPath).
3) Capture all AJAX requests and responses during page loading and modification. This is achieved by overriding the native XMLHTTPRequest JavaScript object implementation provided by the web browser, and adding sniffing functionality to the ”send” method and the ”onreadystatechange” event handler.
4) For each AJAX request, compare the DOM trees before and after the request is completed. Mark all DOM elements that were added or modified as probably originating from the AJAX request (although they may also have been created or modified by JavaScript code or by the browser itself).
5) For each element collected in the previous stage, extract its textual content and check whether the content indeed appears in the incoming AJAX response. If it does, save the URL of the HTTP message containing the data and the location of the data within the message.
6) Compare the resulting DOM after the page has been loaded with the initial DOM. Define as ”locally modified” any elements that exist in the resulting DOM, that do not exist in the original DOM and did not receive their values from AJAX requests.

Once a UI element is selected by the user, the relevant message is extracted from the map and a script for masking that element can be easily created

2. Security barriers Naturally, the rule authoring tool and the target application to mask are installed on distinct application servers or possibly even in a different domains. Thus the browser's same-origin security policy restrictions [33] prevented us from pursuing the naïve approach of presenting the application to be masked in its own frame within a larger rule-authoring tool page and intercepting its mouse events to generate rules according to the selection.

A few options are available for overcoming this challenge, including implementing the rule-authoring tool as a browser add-on, thus possibly requiring re-coding for each supported

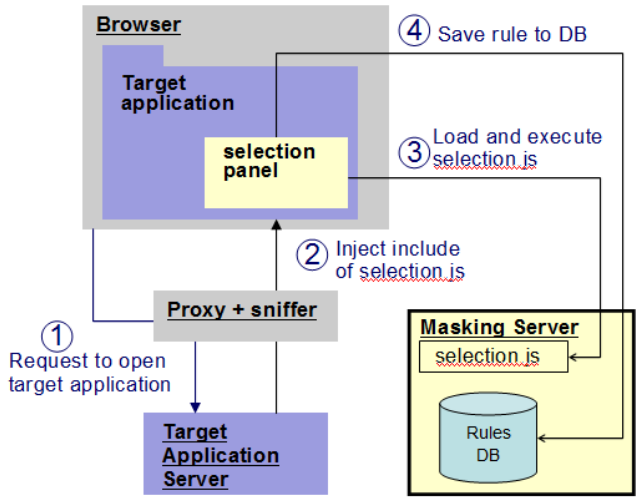

Fig. 5. Cross-origin solution. Target application and selection panel are colored accoding to application server domain.

browser type, or falling back on a stand-alone tool. Both solutions require installation of software on the author's machine.

We chose an alternative approach that leverages the architecture that is already in place for the rule enforcement and does not require any additional installation. As shown in Figure 2, the network is configured to send all communications between the application server and the client browsers through a proxy server, routing them to the sniffer component. When the application is navigated for the purpose of visual rule-authoring, the same sniffer component modifies each application page by injecting a pointer to an external JavaScript file with our implementation of the rule-authoring tool. When this modified page is later processed by the author's browser, the rule authoring code is loaded and executed, as shown in Figure 5. Specifically, it draws the panel shown in Figure 4 and registers mouse-event listeners that will serve for automatic rule generation. No same-origin restrictions exist for loading external JavaScript files, hence the visual rule-authoring tool can analyze and create rules for web applications even if they are deployed on a different application server than the tool itself. As a result, our rule-authoring tool does not require any changes to the application, to the server where it is deployed, or to the authors' machines.

In conclusion, despite the fact that presentation-layer rules are inevitably complicated to create, the visual rule-authoring tool enables a simple rule-authoring mechanism. Rule authors merely need to navigate the application to select a context on the presentation layer, while behind the scenes a complex machine-readable masking rule, to be run during enforcement, is created.

The current implementation of the visual rule-authoring tool can automatically create fully functional masking scripts for table columns with data coming within an HTML page or from JSON-formatted AJAX responses. Masking script creation is also supported for labeled fields coming within HTML pages. Moreover, any area on the screen can be selected and any text elements within that area that come from the original HTML page will be masked.

IV. Comparison with other approaches

In this section we argue that the choice of a screen-based rule-authoring approach coupled with rule enforcement at the network level results in a system that is both usable and powerful. We discuss performance issues in Section V.

We evaluate our system by analyzing its advantages and shortcomings based on several criteria:

A. Rule strength and granularity

A commonly used masking language uses only content-based rules. Content-based rules can be defined in several ways: by a regular expression that defines what text to mask; by using more advanced text analytics tools that utilize Natural Language Processing (NLP); or by other classification methods. Another alternative masking technique is to apply the masking on the application's data layer, i.e., mask the data in the underlying database. We start by comparing the strength and granularity of our context-based masking rules with the content-based and data layer alternatives, as summarized in Figure 6.

A masking system's effectiveness is primarily measured by how powerful the rule language is and the ***masking granularity*** it offers. This measure assesses the capability of a rule author using the system to create a rule that is specific enough to mask all the items of his/her intention, but leave all other items untouched.

An additional metric to evaluate a policy and rule language is ***logical rule coverage*** which assesses the ability to describe a rule by its logical content, e.g., declare a rule like "mask emails of patients". The term ***visual rule coverage*** describes the ability to mask all or part of the elements in a given area of the screen, whereas ***visual screen context*** refers to the ability to create rules in the context of the presentation layer.

Content-based rules have inferior ***masking granularity*** than context-based rules. If, for example, there is a need for a rule to mask phone numbers, a content-based rule can be defined using one or more regular expressions (one for each possible format), whereas in a context-based rule system we need to define a rule for each of the places where sensitive phone numbers appear. However, the content-based rule will always mask all phone numbers in the application. It does not enable, for example, masking only patient phone numbers and not physician phone numbers. Moreover, in cases where a phone number may simply appear as 9 consecutive digits, the regular expression will also mask other 9-digit numbers that are not phone numbers. In other cases, where the texts to mask are not easily represented as regular expressions (such as names and addresses), a regular expression will not suffice and more sophisticated techniques may be required.

Rules defined at the data layer have the advantage that any data item that requires masking is specified for masking only once, regardless of whether it appears on several different pages in the web application or has several formats. For example, if the goal is to mask all customer phone numbers, this can be done by specifying a single logical rule on the relevant database column, thus giving good ***logical rule coverage***, whereas in our approach it could require several rules. However, visually defining a rule at the presentation layer has greater strength in terms of granularity of the context. A data item in a table can appear in two different contexts, one that should be masked and one that should not. An example of this is masking customers' phone numbers. These numbers can appear in two contexts in the application: (1) the page where all customer details are displayed, an instance that requires masking to maintain customer privacy; (2) the order page, where the phone number is specified as a contact method for shipment. Masking this instance is not needed and could even cause problems in distribution. In this example, two phone number instances are extracted from the same table, but given the context, only one needs to be masked.

Comparison Point	DB based	Content Based	Context Based
Masking granularity	+	0	++
Logical rule coverage	++	+	+
Visual rule coverage	0	0	++
Visual screen context	0	0	++

Fig. 6. Comparison of the strength of the different rule authoring mechanisms
A feature is marked as 0 if it is not supported at all, + if it is somewhat supported, and ++ if it is fully supported.

Furthermore the ***visual rule coverage*** will be better in a context-based rule system. A rule author may want to mask all items in a given area of the screen. Since there may not be any correlation in the format or source table at the back-end for these to appear adjacent on the presentation layer, this would be extremely difficult using the data layer or content-based approaches.

B. Rule enforcement mechanisms

Three distinct rule enforcement methods are analyzed: (1) enforcement performed at the back-end of the application, e.g., at the database level; (2) enforcement done at the network level, as in our approach; and (3) enforcement done at the presentation-layer, e.g., the OCR-based approach [8]. This comparison is summarized in the table in Figure 7.

Masking an application can have detrimental results on the proper functioning of the application. Thus, maintaining ***application integrity*** is a primary concern that must be addressed when enforcing masking rules. Of the three enforcement mechanisms listed above, masking at the presentation layer is the safest. The masking is done "on screen", so only the user is aware of it, while the application is unaffected. Masking done in the database has potentially the worst repercussions, since illegal or missing values can result in a "breaking" of the application. When enforcement is performed at the network level, masking is executed on a proxy that resides between the application server and the client-side browser. Thus, the server-side application is not affected by the masking. However, value validation or sorting that is performed on the client-side, e.g., using Javascript, may be compromised.

The ***impact of screen complexity*** on the enforcement of masking rules is reversed. While database masking is independent of the complexity of the screens of an application, masking at the presentation layer is directly correlated to screen complexity. Overlapping or partially visible windows pose a

Comparison Point	DB Enforcement	Network Enforcement	Presentation Layer Enforcement
Application integrity	0	+	++
Role based masking	0	++	++
Impact of screen complexity	++	+	0

Fig. 7. Comparison of the different enforcement mechanisms
A feature is marked as 0 if it is not supported at all, + if it is somewhat supported, and ++ if it is fully supported.

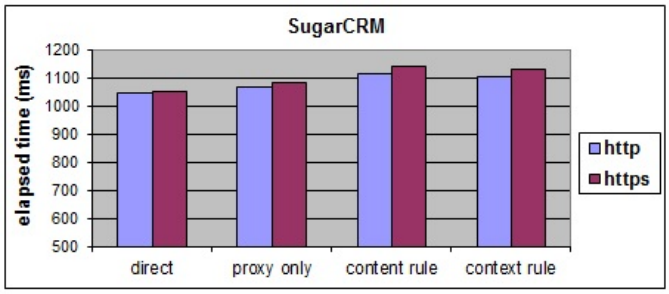

Fig. 8. Masking performance impact

significant challenge for the enforcement of masking rules at the presentation layer. Network-based masking enforcement is somewhat affected by application complexity, but since the proxy resides on the network, all passing traffic is available for scrutiny and masking, and can therefore be ”seen” by the masking engine.

Role-based masking is problematic when masking at the back-end as the application must extract the values from different versions of the database, depending on the user accessing the application. At the network and presentation-layer, this is not a problem, as these solutions are located after the application server and can use existing session information to achieve role-based masking.

V. Performance Results

In this section we show that the runtime performance impact of our system is negligible, and without any meaningful effect on the users' experience. We compared performance results only to content-based rules as it is supported by our architecture and can be fairly compared to context-based rules. The other alternatives were either not available for us or could not be compared to our solution. The OCR-based solution [8] is implemented only for Mainframe applications, and the database level enforcement cannot mask at the same granularity (e.g, cannot mask a single labeled field). We measured the performance of our system using two different applications:

1) SugarCRM - an Open Source customer relationship management application, containing mainly pure HTML pages.
2) Report System - a proprietary application that enables building and viewing reports generated from audit logs. Most pages in this application are composed of one main HTML document, a number of messages in JSON format that bring the data to display, and additional scripts, images and style sheets.

The performance measurements were made using a Linux Redhat server as the masking server, with 2 Intel Xeon 2.5 GHz processors, 24 cores and 20GB of memory.

Figure 8 shows the elapsed time between sending a request and receiving the corresponding response at the client browser for a 300KB HTML page. The same page was requested in four different setups: (1) direct access between the client and server (without any extra layer between them); (2) using a proxy layer between the client and server, but no masking; (3) content-based masking is performed using a regular expression rule; and (4) context-based masking is performed using a rule defined with the visual rule-authoring tool. This test was performed once using the HTTP protocol and once using the secure HTTPS protocol.

Figure 8 shows that adding the additional proxy layer between the client and server adds 2.2% to the overall response time, even with no masking taking place. Discounting the delay due to the proxy, the masking process itself only adds 3.4-4.6% to the response time. When comparing between content- and context-based rules, we can see that there is no significant difference in their processing times, with a slight advantage to the context-based rule (5.7% overhead versus 6.9%, when compared to direct access). This is attributable to the time taken for the message to be parsed and all text nodes to be extracted before the regular expression can be applied, whereas the context-based rule indicates the exact place to mask. The HTTPS protocol entails a higher overhead in all setups due to the fact that the proxy layer introduces an encryption and decryption process in addition to the one performed on the server and client. Therefore, the overhead of adding the masking layer is higher (7.4%-8.5% overhead) when compared to direct access.

Figure 9 shows the effect of the number of masked elements on the performance overhead. For both applications we defined rules that mask a table column and in each run we increased the number of rows in the table. In the SugarCRM application the measurements were performed on a table where the data arrived in an HTML message whereas in the Report System application the data arrived in JSON format. The graphs show that the overhead introduced by the masking layer increases with the message size as expected. In SugarCRM it can again be seen that the overhead incurred by the context-based rule is slightly lower than that of the content-based one. In the Report System we see a small advantage to the content-based rule. This can be explained by the fact that JSON messages are much smaller than HTML messages so extracting the text nodes and checking regular expressions in JSON is quite fast. On the other hand, the scripts that mask a column are more complicated in JSON than in HTML.

To test the performance of the system under load, we used the open-source tool Apache JMeter™ [34], which enables running a number of concurrent threads sending requests to the application server. Figure 10 shows the effect of the number of concurrent threads on the performance overhead. In both applications we increased the number of client threads requesting the same page at the same time and measured the elapsed time to process a message. In SugarCRM the message

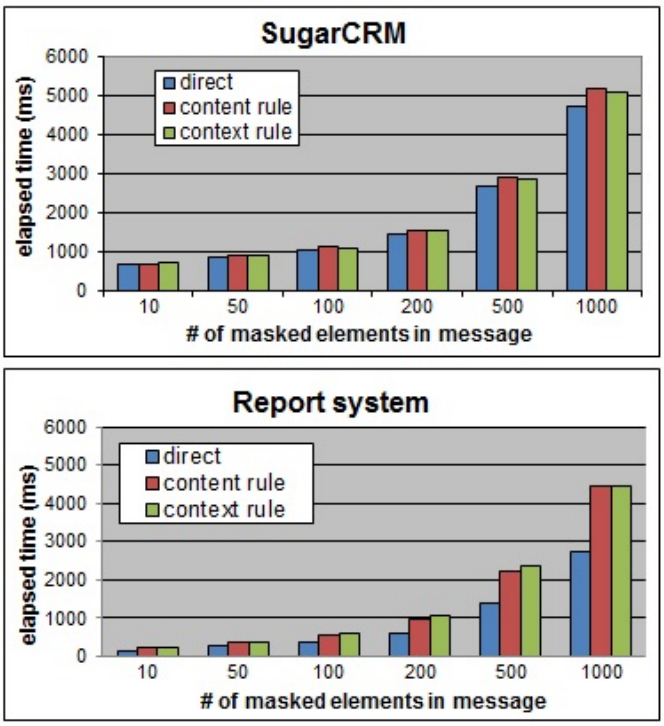

Fig. 9. Influence of message size and number of masked elements on masking overhead

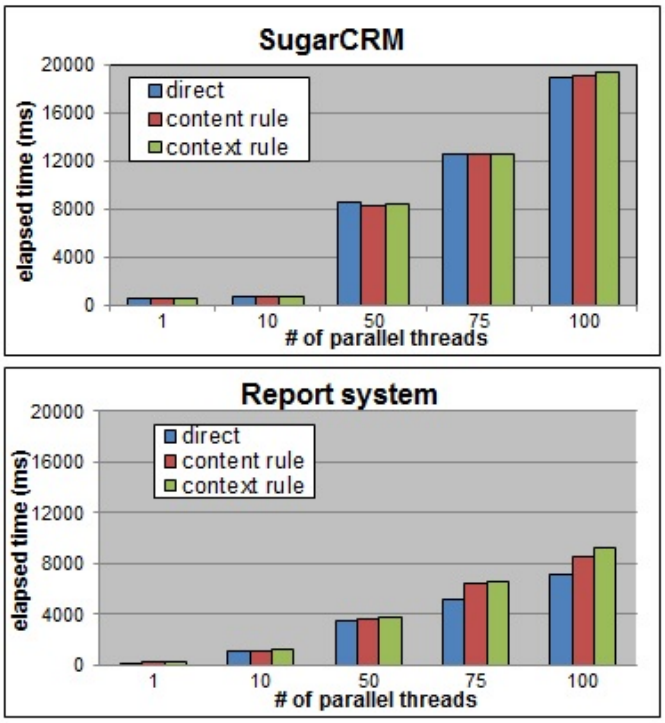

Fig. 10. Influence of number of client threads on masking overhead

was in HTML format and in the Report System it was in JSON. In SugarCRM there is a small impact of the number of threads on the overhead introduced by the masking layer, which ranges from 4.1% with a single client thread to 2.1% with 100 client threads. There is even a slight decrease in the masking overhead as the thread number increases, which we attribute to the fact that the application server itself did not cope well with the higher numbers of threads. In the Report System, for a small number of threads (up to 50) the impact of masking is relatively low, 4% to 10%. When the number of concurrent threads increases, the impact of the masking increases as well.

In both applications the performance of the context-based rule degraded with the increase in the of number of threads due to a limitation in the Javascript engine that does not enable the same script to be executed concurrently in different threads. Therefore, although most of the system supports multi-threading, part of the rule enforcement is performed serially.

The JMeter tool simulates multiple users working with a web application simultaneously by running a number of concurrent threads, each one sending sequential requests to the application. However, in reality, application users behave quite differently. After loading a page one typically needs some time to read the information the page provides, make a decision as to what to do next, or input data to a form. This time may differ significantly, not only between users, but also between applications, and could range from a few seconds to even minutes. Thus, users will not all load new pages at exactly the same time. Therefore, the number of concurrent users that can be supported by an application is typically much larger than the number of concurrent threads accessing the application at a given time. Relying on the fact that users will typically spend 1-2 minutes on each page, it is estimated that 100 concurent threads is simulating approximately 2000-2400 concurrent users.

VI. Summary and Future Work

This work focuses on masking sensitive data in web applications. We show that a hybrid approach, combining context-based rule creation at the presentation level with enforcement at the network level, enables a powerful and flexible mechanism for masking sensitive information. The rule-authoring tool supports the creation of complex masking rules while keeping the rule authoring process easy and straight-forward. Our system enables rule authors to define masking rules in a simple and intuitive manner while navigating the target application and clicking on the sensitive areas. These rules are then transformed into machine-readable instructions to be enforced at runtime. This system is planned to be part of an IBM product.

Our work can be extended in the following ways:

- Enhance the existing protocol-based implementation, that currently supports the HTTP and HTTPS network protocols and the common payload formats HTML, JSON, and XML. Our solution can be easily extended to support additional protocols and formats.
- Extend the visual rule-authoring tool to handle additional application types (e.g., based on Dojo[1], particularly tables rendered using the Dojo Grid widget) and formats (e.g., XML), and to support additional UI constructs and more complex context-based rules involving more than one visual element.

[1] http://dojotoolkit.org/

- Improve the script-based mechanism to be more flexible to changes in the document structure, both to cover cases where two pages may look identical to the rule-author but have slightly different underlying structures, and to prevent malicious users from exploiting XSS vulnerabilities to change the document structure and avoid masking.
- Improve the request reconstruction mechanism to prevent malicious users from revealing sensitive information by copying masked values to other fields in the request.
- Perform URL canonicalization to ensure that a page will be masked regardless of changes to the URL (i.e., use of non-canonical forms).
- Add a verification component to analyze the masked application offline and verify that all sensitive information was indeed masked.

Our approach has many advantages but it also has some limitations. Sensitive content may appear within an object (e.g. Flash, Java Applets, or simply images) transmitted in binary format and not as part of the textual network protocol, or generated on the client-side. This cannot be masked by our system. We experimented with combining visual and protocol information during masking-rules authoring; however, we believe that a holistic solution should include such a combination at the enforcement point as well.

In addition, applications that perform client side validation (e.g., of formats or ranges) may generate unjustified warnings on masked or removed values. This can be solved by replacing the values using format-preserving methods.

ACKNOWLEDGMENTS

We would like to thank the IBM Guardium development team for enabling and contributing to the development of the network-based enforcement engine. We thank Irit Cohen who contributed to the development of the visual rule-authoring tool, and Ming Dong who helped in gathering the system's performance measurements.

REFERENCES

[1] "Directive 95/46/EC of the European Parliament and of the Council," http://ec.europa.eu/justice/data-protection/index_en.htm, 1995.

[2] D. Binkley, "Source code analysis: A road map," in *Future of Software Engineering*, May 2007, pp. 104 – 119.

[3] T. Elrad, R. E. Filman, and A. Bader, "Aspect-oriented programming: Introduction," *Communications of the ACM*, vol. 44, no. 10, October 2001.

[4] S. R. M. Oliveira and O. R. Zaane, "An efficient one-scan sanitization for improving the balance between privacy and knowledge discovery," Department of Computing Science, University of Alberta, Canada, Tech. Rep., 2003.

[5] A. D. Brucker and H. Petritsch, "Extending access control models with break-glass," in *Proc. 14th ACM symposium on Access control models and technologies*, ser. SACMAT '09. ACM, 2009, pp. 197–206.

[6] "XML path language (XPath) 2.0," W3C Recommendation http://www.w3.org/TR/2007/REC-xpath20-20070123/, World Wide Web Consortium, January 2007.

[7] S. Ansari, S. Rajeev, and H. Chandrashekar, "Packet sniffing: a brief introduction," *Potentials, IEEE*, vol. 21, no. 5, pp. 17–19, January 2002.

[8] S. Porat, B. Carmeli, T. Domany, T. Drory, A. Geva, and A. Tarem, "Dynamic masking of application displays using OCR technologies," *IBM Journal of Research and Development*, vol. 53, no. 6, 2009.

[9] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *Proc. 2010 IEEE Symposium on Security and Privacy*, 2010, pp. 332–345.

[10] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *Proc. 2006 IEEE Symposium on Security and Privacy*, 2006, pp. 258 – 263.

[11] M. Cova, V. Felmetsger, D. Balzarotti, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *Proc. 2008 IEEE Symposium on Security and Privacy*, 2008, pp. 387 – 401.

[12] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *Proc. 2010 IEEE Symposium on Security and Privacy*, 2010, pp. 191 – 206.

[13] S. Chong, K. Vikram, and A. C. Myers, "SIF: Enforcing confidentiality and integrity in web applications," in *Proc. 16th USENIX Security Symposium*, 2007, pp. 1 – 16.

[14] V. Ganapathy, T. Jaeger, and S. Jha, "Retrofitting legacy code for authorization policy enforcement," in *Proc. 2006 IEEE Symposium on Security and Privacy*, 2006, pp. 214 – 229.

[15] S. Lodha and S. Sundaram, "Data privacy," in *Proc. 2nd World TCS Technical Architects' Conference*, Hyderabad, India, 2005.

[16] X.-B. Li and L. Motiwalla, "Protecting patient privacy with data masking," in *Proc. 4th Annual AIS SIGSEC Workshop on Information Security and Privacy*, Phoenix, AZ, USA, 2009, pp. 214 – 229.

[17] "IBM Infosphere Optim," http://www.optimsolution.com, IBM.

[18] "Oracle Data Masking," http://www.oracle.com/us/products/database/data-masking-161222.html, Oracle.

[19] "Camouflage Data Masking," http://www.datamasking.com/solutions/products/datam Camouflage.

[20] "Voltage SecureData Masking," http://www.voltage.com/products/data-masking.htm, Voltage Security.

[21] "Verdasys Digital Guardian application logging and masking module," http://www.verdasys.com/pdf/Digital_Guardian_ALM_DS.pdf, Verdasys.

[22] "Intellinx Enterprise Fraud Detection & Prevention," http://www.intellinx-sw.com/, Intellinx.

[23] "IBM Infosphere Guardium Data Security," http://www-01.ibm.com/software/data/guardium/, IBM.

[24] "Check Point DLP Software Blade," http://www.checkpoint.com/products/downloads/datasheets/DLP-software-blade-datasheet.pdf, Check Point.

[25] B. Liver and K. Tice, "Privacy application infrastructure: Confidential data masking," in *Proc. 2009 IEEE Conference on Commerce and Enterprise Computing*, 2009, pp. 324 – 332.

[26] "Riverbed Stingray Traffic Manager," http://www.riverbed.com/products-solutions/products/application-delivery-stingray/, Riverbed.

[27] J. Elson and A. Cerpa, "Internet content adaptation protocol (ICAP)," RFC 3507 http://tools.ietf.org/html/rfc3507, April 2003.

[28] "Squid: Optimising web delivery," http://www.squid-cache.org.

[29] "The c-icap project," http://c-icap.sourceforge.net.

[30] "Libxml2," http://www.xmlsoft.org/.

[31] "Jansson," http://www.digip.org/jansson/.

[32] "Mozilla Spidermonkey," https://developer.mozilla.org/en-US/docs/SpiderMonkey, Mozilla Developer Network.

[33] J. Ruderman, "The same origin policy," http://www.mozilla.org/projects/security/components/same-origin.html, August 2001.

[34] "Apache JMeter," http://jmeter.apache.org/, The Apache Software Foundation.