Clilets: Web Applications with Secure Client-Side Storage

Clilets: Web Applications with Secure Client-Side Storage

Robert Fischer
and
Margo Seltzer

TR-11-02

Computer Science Group
Harvard University
Cambridge, Massachusetts

Clilets: Web Applications with Secure Client-Side Storage*

Robert Fischer Margo Seltzer
{citibob,margo}@eecs.harvard.edu

Abstract

Today's web applications require that all data be visible to the server. This is a problem in cases, such as a Web Tax service, where the user may not trust the server with the data. We present the Clilet system, a new web application system that allows sensitive data to be stored securely on the client yet still accessed by the web application. The system ensures that this data is not transmitted to the server, even though no trust is shared between client and server. We have built a working prototype.

1 Introduction

Web applications generally process data of some sort: bank account transactions, airline reservations, etc. Whether that data is stored in a database on the server or in cookies on the client, it is visible to the server. Therefore, the user must trust the web application with all data that it processes. If the data originate with the web server, then it is certainly acceptable that the web server has access to it.

However, when those data originate from the user, sharing them with the server may require a level of trust with which the user is not entirely comfortable. It is neither necessary nor desirable that users should be required to reveal personal or private information to a web server simply to use the services provided by that server.[1] This required trust relationship can become problematic in a number of real-world scenarios:

Tax Preparation Software Some tax preparation services provide web applications that process tax forms. In using these "Web Tax" applications, the user must reveal all relevant tax data to the preparation service via the web, thereby trusting the Web Tax service with sensitive financial information. Even if the service's intentions are good, the fact that it has so much personal tax data aggregated in one place makes it a prime target for hackers or other intrusions.

Banking Web Site Consider a bank web site that provides a check register maintenance service for its customers. When the user logs on, the bank provides a record of checks that have cleared (by check number), the date they cleared, and their dollar amount. By summing the cleared checks, the service also computes an account balance.

There might be additional information relevant to the account and necessary to compute its balance, e.g., a record of uncleared checks, a description of each check. The bank cannot provide this information: it generally does not read handwriting on checks, and it cannot know about checks that have not yet cleared. The bank allows users to add this information to their on-line account record, thereby constructing a complete version of the account status. That way, the user does not have to interpret check numbers seen on the screen, and the displayed check register has an up-to-date account balance that includes uncleared checks.

However, the customer might not trust the bank with this "extra" information. For example, a bank user probably does not wish for the bank to know that s/he knowingly wrote a check for more than the current balance, anticipating a deposit to be made before that check clears. Alternately, the user might be involved in an embarrassing activity about which the bank (or others) ought not know: e.g., that the innocuous $1,000 check was for the President's party.

Shared Calendars Web-based calendars allow users to keep track of their schedules and coordinate with others.

*Submitted to 3rd USENIX Symposium on Internet Technologies and Systems (USITS '03)

[1] Note that SSL, the basis for the HTTPS protocol, only protects personal data while it is in transit to the server. It does not provide privacy once the data reaches the server.

The web, by allowing the sharing of information between users, is a perfect medium for this task. However, certain information, meant only for the user's private calendar, should not be shared. For example, a user might not want the boss to know of the planned trip to Vegas on a sick day.

Not trusting centralized web servers with their private appointment data, users will typically keep two calendars: one on the web for public data, and one for private data. They must manually "integrate" the information on the two calendars when considering their schedule. It would be nice if one integrated application could securely handle both sets of data.

Discussion

Today's web application protocols do not provide the required security for sensitive user data. In all three cases above, it is simply a matter of convenience that the server has access to the data; no one has come up with an alternate architecture that would permit the preservation of user data privacy. As a result, the potential of today's web applications is far from realized: savvy users are often not willing to use them due to these very real security concerns.

In order to address these concerns, we have developed a new application-level web architecture and protocol, called the *Clilet System*. As with HTTP/HTML, a standardized browser, called the *Clilet browser*, functions as a client for all applications following the Clilet protocol. However, the Clilet system allows the application developer to specify that certain pieces of data are private. They are stored on the client, generally in a database or other type of persistent storage. The Clilet browser advertises this fact to the user via visual cues (e.g., the color or style of text on the screen).

Although data stored on the client is not revealed to the server, the web application is still allowed to access and process that data. Thus, we have a seeming paradox. The web application must be able to process, read and write client-side data, but the server on which it is running is not allowed access to that data.

We have developed a fully secure, fully functional prototype implementation of the Clilet system, which includes a Clilet browser, a server development framework, and a sample server. In this paper, we will focus on the browser because that is the component requiring new security mechanisms.

The rest of this paper is organized as follows. Section 2 introduces the Clilet protocol and discusses its security requirements. Section 3 describes our prototype implementation. Because it executes mobile code on the client, the Clilet system introduces a number of interesting problems that must be addressed in order to achieve the stated goal of Client data privacy; some are new problems, whereas others have already been solved in the literature. Sections 4 and 5, discuss security mechanisms in the Clilet browser. Finally, we offer possible applications of the Clilet system and its technology in Section 6. We point to directions for further research in Section 7 and conclude in Section 8.

2 Clilet Protocol

The main goal of the Clilet system is to allow web applications to store and process data in the Clilet browser without being able to leak that data to the server. We call this data *private data*.

To users sitting at a Clilet browser, a Clilet protocol session works much like an HTTP/HTML session. Users are presented an HTML page, with which they interact. When a user clicks on an anchor or form submit button, the Clilet browser initiates a round-trip communication with the server, called an *interchange*; the effect of that communication is to display a new HTML page in the browser. Sessions are managed through the use of session keys shared between browser and server.

The workings of the Clilet protocol are similar to HTTP, with one important difference. Rather than returning HTML to the browser, the server returns a *clilet* — a kind of mobile agent that includes data and executable code. The clilet, when run, produces HTML, which is displayed to the user.

Clilets are sent to the Clilet browser in two pieces, the *public clilet segment* and the *private clilet segment*. The public clilet segment generates the bulk of the HTML. The private clilet segment is responsible for reading and writing private data, and for generating HTML that will display that data to the user.

Protecting the privacy of private data is a hard problem because of the unique trust model in which the Clilet system operates. Although the Clilet browser is considered to be trusted, the server and its clilets are completely untrusted. The Clilet browser must allow for untrusted clilets to run and access private data, but not to

collaborate with the server in transmitting that data to the server.

2.1 Clilet Execution

Clilet execution begins at an entry point in the public clilet segment; from that point, a single thread of control passes between public and private clilet segments. Every time control switches from public to private clilet segment, we say the clilet has *switched context*. Context switches are necessary because only the private clilet segment is allowed to access private data, whereas only the public clilet segment is allowed to write certain kinds of HTML.

The two clilet segments collaborate to generate the HTML page that the user will see, successively appending to the end of the output as they run. Clilets are allowed write-only access to the output stream. The resulting HTML therefore consists of an alternating series of segments, called *public HTML segments* and *private HTML segments*; they are written by the public and private clilet segments, respectively.

2.2 Information Flow

Having restricted private data access to the private clilet segment, we must prevent the flow of information from the private clilet segment to all other untrusted entities in our system: that is, to the public clilet segment or to the server. If we can do this, then we can be sure that user data privacy is maintained, even though we have provided a piece of untrusted code — the private clilet segment — access to the private data.

We have taken the obvious step of sandboxing the private clilet segment so it cannot communicate directly with the server. However, there are three other ways the private clilet segment might indirectly transmit data to the server:

Information Flow via Public Clilet Segment: The private clilet segment transmits data to the public clilet segment, which then communicates it to the server.

Information Flow via HTML: The private clilet segment writes data into anchors or form elements, which are then transmitted to the server when the user interacts with the resulting HTML page. For example, the private clilet segment might put private data into an anchor's URL.

Information Flow via User: The clilet tricks the user into manually revealing private data to the server. For example, the Web Tax system might get the user to click on a different anchor depending on that user's income.

Since we do not use static information flow analysis on the clilets [3, 5, 6], we must assume that any and all communication between the private clilet segment and other entities *might* be transmitting information.

The Clilet browser contains sub-systems that prevent these communications. The *Multi-Domain Sandbox* (Section 4) prevents information flow from private to public clilet segment. The *HTML Verifier* (Section 5) prevents information flow to the server via HTML; it prevents, for example, private clilet segments from writing anchors into the HTML output stream.

There is unfortunately no way to prevent information flow via the user. Depending on the application, the subtleties involved could challenge even astute users. For example, the Web Tax system — or even a traditional web site that just serves up tax forms — might gain information about the user's financial state based on the set of forms downloaded.

Any secure system can be compromised unwittingly by its users actions. Users must understand the security model in order to maintain it. Future research could focus on conveying the security model effectively, so that they do not inadevertantly leak private data.

2.3 Other Security Properties

In addition to user data privacy, a number of other security properties most be maintained by any realistic Clilet system. For example, clilets should not be able to initiate worms or viruses. Browsers and servers should have strong ways to authenticate each other. Communication between browser and server should be encrypted to maintain privacy. If the Clilet browser is running on a multi-user system, its data must be kept secure from unauthorized users.

These and other security properties have been extensively studied. We have implemented them, or weak versions of them, only as necessary to demonstrate the viability of user data privacy as a useful and implementable security property. Classical security mechanisms can be "added on" to the Clilet system as needed, in the obvious ways, to produce a system that meets the security

requirements for a specific application. Their omission in this paper does not necessarily indicate a flaw in the idea of user data privacy or in the methods we use to implement it.

2.4 Conclusion

Beginning with a trust model, we built an architecture in which only the (untrusted) private clilet segment is allowed access to private data. From there, we concerned ourselves with preventing the flow of that information from the private clilet segment to other untrusted entities, such as the public clilet segment or the server. We built two client-side mechanisms to prevent these information flows: the *Multi-Domain Sandbox* and the *HTML Verifier*. We also examined information flows enabled by the user, but they are outside the scope of this paper.

We have therefore given good reason that our architecture is sound: that when implemented correctly, it yields a system in which user data privacy is maintained in the face of untrusted servers and clilets. A formal proof of this fact is outside the scope of this paper, but will be presented in the future. Other work will be needed to verify that a particular implementation of the Clilet browser meets the security specifications.

3 Browser Architecture

We described the Clilet protocol and gave reasons for its security in Section 2. Clearly, the interesting part of the system lies in the Clilet browser and how it deals with untrusted clilets. In this section, we describe the architecture of our prototype Clilet browser.

3.1 Web Browser Front-End

Ideally, we would develop a Clilet browser as a self-contained application program to be run on a client machine. However, the Clilet system currently faces the chicken-and-egg problem of no installed browsers.

Since the Clilet system user experience is so similar to the HTTP/HTML user experience, we sought to solve this problem by using a standard web browser as the user interface for our Clilet browser. In this way, Clilet applications could be used from any computer with an installed web browser.

Since a stock web browser cannot speak the Clilet protocol, something needs to translate between HTTP/HTML and the Clilet protocol, allowing a standard web browser to be used as a front-end to Clilet applications. This is accomplished through a piece of middleware, called the *Glue*, that functions as a Clilet system client and an HTTP server (see Figure 1). Note that our implementation of the Clilet browser specified by the Clilet protocol consists of the Glue and stock web browser coupled together.

The use of the stock web browser as a front-end ultimately saved us much effort and provided great flexibility. However, its use also introduced some problems that are artifacts of this approach. We summarize those problems here:

- Presuming that the Glue is receiving HTTP connections from anywhere, it must authenticate users. This can be done as in any standard web application. Alternately, if it is running on a single user system, the Glue can reject all HTTP requests from outside the local host.
- If the Glue is willing to partake in HTTP sessions outside the local host, those sessions must be encrypted via SSL to maintain user data privacy.
- The user must trust the computer on which the Glue runs.
- The Clilet system uses a variant of HTML that is somewhat different from that used by a stock web browser (see Section 5). The Glue must translate Clilet HTML into HTML suitable for display by the stock browser. It must also reject HTML that tries to invoke features, such as JavaScript, that are implemented by the stock web browser but have been purposefully removed from Clilet HTML.

3.2 Java

As with other mobile agent systems, the Clilet system requires a standard language in which to code mobile agents, a standard way to transport those agents between computers, and way to safely run those agents in a controlled environment. We chose to use the Java Programming Language [2] for the Clilet system because it provides all three of these mechanisms in an easy-to-use but flexible and configurable fashion. The ideas behind the Clilet system could be implemented outside the Java environment as well.

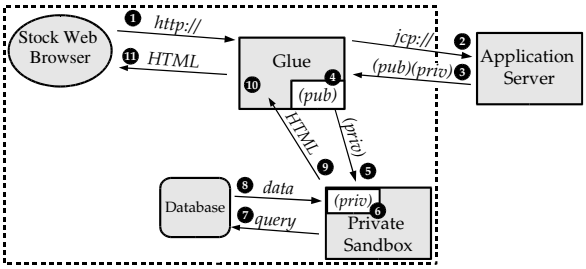

Figure 1: The major components of the prototype Clilet browser, and how they interact on a client-server interchange. Numbers indicate the order in which interchanges take place. All components within the dotted line are part of the Clilet browser.

3.3 Prototype Interchange

Our prototype Clilet browser actually consists of three client-side processes communicating with each other: the Glue, the Private sandbox and the stock web browser. All three of these components are part of the Clilet browser. All three must be trusted by the user, just as the user must trust a stock web browser when using HTTP. We assume that all three are running on the same computer; if not, trusted (encrypted) channels must be used to communicate between them.

With the architecture of our prototype Clilet browser in mind, we present again, in more detail, the steps that take place in a Clilet protocol interchange (See Figure 1):

1. The stock web browser sends an HTTP request to the Glue.

2. The Glue translates the HTTP request into the Clilet protocol and forwards it to the Clilet server. By acting as an intermediary between browser and server, the Glue allows use of the Clilet protocol from any standard web browser. This eliminates the need to build and deploy a special browser for the Clilet protocol.

3. The server processes the request. In processing, it could do a number of things, such as access a local database, connect to a legacy mainframe, etc. When it is ready, it returns public and private clilet segments to the Glue.

4. The Glue runs the public clilet segment in an in-process public sandbox; HTML output is buffered in the Glue. Whenever it needs to display private data, the public segment (seemingly) makes a function call to the private clilet segment.

5. Regaining control, the Glue suspends execution of the public clilet segment, opens a connection to the out-of-process private sandbox, and sends it the private clilet segment.

6. The private sandbox runs the private clilet segment. HTML output is buffered in the private sandbox.

7. In the process of execution, the private clilet segment accesses the private data store. This can happen more than once. Access is provided through the private sandbox server.

8. The private data store returns private data to the private clilet segment.

9. When the private clilet segment is finished, its output is returned to the Glue. The Glue incorporates the private HTML segment with HTML into the on-going HTML output buffer. Annotations are added to remember which HTML came from private clilet segments. The Glue then returns control to the public clilet segment.

10. When the public clilet segment is finished, the Glue passes the resulting HTML stream through the in-process HTML Verifier. This ensures that the HTML is not being used to leak private data to the server; specific leakage issues are addressed in Section 5.

11. HTML that passes the verifier is sent to the browser; HTML that fails the verifier results in an error message being sent to the browser.

Having given an overview of the Clilet system specification as well as our implementation of it, we now give the details of the browser security mechanisms used to run clilets: the Multi-Domain Sandbox and the HTML Verifier.

4 Multi-Domain Sandbox

As mentioned above, the multi-domain sandbox prevents information flow from the private to the public clilet segment.

This is done by building a multi-level system, along the lines of the Bell-LaPadula model [1], in which the public and private clilet segments run. The public and private clilet segments therefore run in different security domains, called the *public sandbox* and the *private sandbox*.

The private clilet segment runs in a different security domain from the public clilet, with no shared global state between the two. It performs the required task and returns control to the public clilet segment. Its output was written to the HTML output stream, and is *not* made available to the public clilet segment.

All function calls to the private clilet segment are forced by the multi-domain sandbox to return a void data type to the public clilet segment. In this way, the required task is accomplished, yet the public clilet segment gains no information about the contents of the private data. Data is allowed to flow from the public to the private clilet segments, but not the other way.

4.1 Information Flows

The typical malicious private clilet segment might try a variety of ways to pass data back to the public clilet segment. We classify these methods as either *Direct Channels*, in which the private clilet segment tries to use a direct communication channel to the public segment — or *Covert Channels*, in which the private segment uses a shared mechanism not normally intended for communication. We have addressed these concerns in our multi-domain sandbox:

4.1.1 Direct Channels

The public clilet segment calls the private clilet segment via a synchronous function call. The simplest direct channel would be for the private clilet segment to pass back a return value. This is prevented by design: by definition, the public clilet segment receives a void in return. The Java System would not allow a private segment to return a value.

The private clilet segment might also try to throw an exception, which will be propagated up the call stack. This is prevented because the Clilet browser catches all exceptions from private clilet segments and does *not* propagate information about them to the public clilet segment.

4.1.2 Covert Channels

The clilet might try to use a variety of more subtle covert channels:

- Pass data through the HTML output stream. The private and public clilet segments are allowed to write data to the HTML output stream; however, since neither segment is allowed any information about the state of that stream, the stream cannot be used as a covert channel.
- Pass data through covert timing channels. The private clilet segment could, for example, vary its running time. This is prevented by eliminating all concurrency, including access to the system clock [9, 8].[2]
- Pass data through the use of shared system resources. For example, a private clilet segment could use vast quantities of memory, forcing the public clilet segment to page. This is not a viable communication channel because the public clilet segment cannot detect this kind of system activity, or even the timing glitches it might cause. In any case, Java's garbage collection makes memory system unpredictable.
- Pass data through shared system state. This is not possible in our implementation because public and private clilet segments are run in different JVMs that do not share state.

4.2 Implementation

Having conceived of the multi-domain sandbox, we had to face the problem of its implementation in Java. It is clearly possible to build one [10]. However, Java was not designed with an eye towards multi-level security. Any shared state between the public and private sandboxes can be used to build a covert channel.

Seeing no obvious easy way to guarantee lack of shared state between classes within one Java Virtual Machine (JVM), we found we could instead implement our multi-domain sandbox in two separate JVM's.

In the Clilet system, the Glue is responsible for running public clilet segments. Private clilet segments are run by the *Private Sandbox* in a separate JVM on the

[2] Preventing access to the system clock is not possible using the standard Java system, but it was not difficult to modify the Java System source code to allow this.

client machine. When it wishes to run a private Clilet segment, the Glue sends the appropriate code and data to the Private server. The Private sandbox runs the segment under appropriate security constraints and returns the HTML that resulted. It is up to the Glue to further process that HTML properly.

By using separate JVMs for the public and private clilet segments, covert channels are eliminated with minimal programming effort, at the cost of efficiency. How to implement a multi-domain sandbox in one JVM, which would be orders of magnitude more efficient, remains an open but probably solvable question.

5 HTML Verifier

The writing of HTML can result in communication with the server: for example, data written into an anchor or an inline image tag is transmitted to the server. Therefore, the Clilet browser must be careful about what HTML it allows the clilet to write. The HTML Verifier accomplishes this task, preventing private clilet segments from leaking data to the server via the HTML output stream.

The HTML Verifier examines the HTML stream after it is written by the clilet and before it is displayed on the user's screen. Upon examining the clilet's HTML output, it chooses one of two options. It can accept the HTML, displaying it in the browser's window and allowing the user to interact with it thus enabling further interaction with the server. Alternately, the HTML Verifier can reject the HTML, displaying an error message to the user instead. In this section, we describe how the HTML Verifier works.

The HTML Verifier's work is divided into two tasks. First, it must ensure that all HTML written by the clilet strictly conforms to a well-defined HTML specification, *Clilet HTML*. Clilet HTML is an HTML subset that removes dangerous features such as JavaScript.

Then, the HTML Verifier is left to focus on the more subtle forms of information flow. Any Clilet HTML features that can cause data to be communicated between clilet and server must be examined. We found that these features fall into a few categories: *Anchors and Forms*, *Context Switches*, *Form Elements*, and *Secondary Interchanges*.

In our prototype, the HTML Verifier is also responsible for transforming Clilet HTML written by the clilet into standard HTML to be displayed by the stock web browser. Like the Glue, this is an artifact of our choice to use a stock web browser as our front-end, not a fundamental characteristic of the Clilet architecture.

5.1 Clilet HTML

Since HTML carries security ramifications for us, we must clearly define the HTML variant, called *Clilet HTML*, used by the Clilet system. We start with XHTML 1.1, the latest version of the HTML standard (as of June 2001). XHTML 1.1 is HTML reformulated in XML, rather than SGML in which its predecessor was formulated. From a practical point of view, one can think of XHTML 1.1 as a very clean and well-defined HTML specification.

To obtain our Clilet HTML, we first had to remove portions of the language that should never be used by the Clilet system. JavaScript, for example, could easily be used to subvert user data privacy. We call the resulting HTML subset *Clilet HTML*; it is this HTML that clilets must produce.

Whether or not we write the Clilet browser from scratch, it is necessary to figure out which parts of HTML to remove to form Clilet HTML. Removing features from XHTML 1.1 was easy because it was made modular for this purpose.

We systematically examined modules, removing all modules that could cause security problems. This included modules that could invoke arbitrary plug-ins, modules that allow for arbitrary client-side executable code such as JavaScript and applets, deprecated modules which were included in XHTML 1.1 only for backwards compatibility, and modules for which we had never seen an implementation in a working stock web browser.

We ultimately removed the following modules: *Image*[3], *Ruby*, *Scripting* and *Param*[4], *Embedded Object*, and *Legacy Markup*.

Our prototype implementation ensures that clilet output conforms to the Clilet HTML specification. It does this by running the page through a validating XML parser.

5.2 Anchors and Forms

Anchors and forms contain information that is potentially transmitted to the server. They must therefore be preven-

[3]The XHTML 1.1 Image Module does not define the `<img>` tag, but rather provides a more general framework by which arbitrary plug-ins may be invoked; we have therefore not removed inline image functionality.

[4]Used for parameters to applets and similar executable objects.

ted from containing, or relying upon, private data. The HTML Verifier does this by preventing the use of the <a> and <form> tags in private HTML segments. Clilet HTML contains many features that act like anchors: clickable images, for example. These are also prohibited in private HTML segments, the same as anchors.

5.3 Context Switches

Certain HTML elements — anchors, for example — only make sense if an entire sub-tree falls entirely in a public or private HTML segment. One could imagine a clilet that could subvert user data privacy by writing an anchor's begin and end tags in the public HTML segment, but writing the text displayed to the user in the private HTML segment.

For example, the clilet segments might collaborate to write a series of anchors, each one with the begin and end tags written by the public segment, but the text to be displayed written by the private segment. The private segment could use this setup to build alternate versions of the HTML page that look the same to the user but send different information to the server, depending on the contents of the private data.

The key issue here is that the <a> tag generates a single user interface element on-screen but requires more than one node in the XML parse tree to describe. HTML Verification makes sense only if the entire *subtree* describing the on-screen element is public or private. This property is ensured by prohibiting context switches while an <a> tag is open; i.e. after an open tag but before a close tag.

The list of tags in Clilet HTML with this property is: <a>, <link>, <area>, <map>, <input>, <button>, <option>, <optgroup>, <select> and <textarea>. The HTML Verifier prohibits context switches while any of them remain open.

5.4 Form Elements

Although form tags are not allowed in private HTML segments, it would be too restrictive to prohibit form elements as well. Instead, the Clilet browser distinguishes between *public form elements* — those occurring in a public HTML segment — and *private form elements*.

Data entered into public form elements are sent to the server upon initiation of an interchange, the same as in a classical web application. For this reason, the Clilet browser renders public form elements distinctively, to warn the user that data entered into them is not private and will be disclosed to the server. In our prototype implementation, public form elements are rendered in a distinctive red color.[5]

Data entered into private form elements is NOT sent to the server; rather, they are made available to the private clilet segment that the server returns in the up-coming interchange. In this way, the Clilet application is able to query the user for input and store it locally in the Clilet browser.

Private form elements are rendered by the browser in any style, as requested in the private HTML segment. It is not a major security problem if they look like public form elements to the user, since this will only spur the user on to greater caution than is necessary.

The work of the HTML Verifier with form elements is therefore to recognize the difference between public and private form elements and to make sure the Clilet browser handles them appropriately. In our prototype, the HTML Verifier does this by renaming form elements so the Glue will know, at a glance, which ones are public and which ones are private. It also transforms the HTML to ensure that public form elements show up in red.

5.5 Secondary Interchanges

An interchange in a classical web application consists of more than just one exchange of information. Frequently, HTTP returned by the server will require that other files be downloaded in order to display properly; for example, <img> tags are used to include in-line images. The browser is required to download the specified files, or find them in a local cache, before displaying the page. We call the interchanges triggered by these requests for images and other such objects *secondary interchanges*, to distinguish them from the primary interchange that returns the basic HTML to the browser.

Secondary interchanges would offer a great opportunity for private clilet segments to leak private data. For example, a private clilet segment could encode private data into the URL of an <img> tag. Therefore, private clilet segments are not allowed to initiate secondary interchanges in the Clilet system.

Prohibiting all the functionality gained from secondary interchanges would be draconian: a private clilet seg-

[5] A production version would require additional visual cues as well, since it is bad user interface design to rely on color alone.

ment might legitimately wish to include, for example, a custom icon represented by a GIF image in its output. Therefore, we offer cache tags as a way that the benefits of secondary interchanges may be realized without the security risk.

5.5.1 Cache Tags

We add a new tag, `<cache>`, to our HTML; the HTML Verifier ensures it is only allowed in public HTML segments. Each `<cache>` tag references one URL. Before displaying the HTML page, the Clilet browser downloads all URLs referenced by the `<cache>` tags, if they are not already cached locally. This process provides no information to the server about the private data because `<cache>` tags are written only by the public clilet segment.

Now consider the HTML rendering behavior of the Clilet browser. When it encounters a tag in a private HTML segment that would normally trigger a secondary interchange, it cannot forward the request to the server. Instead, it compares the requested URL to the list of URLs downloaded by the page's `<cache>` tags. If the desired URL is present, the Clilet browser displays it from the cache. If not, the Clilet browser produces an error: a broken image icon, for example. In this way, private segment secondary interchanges are prevented.

In our prototype, we implemented this caching behavior in the Glue and the HTML Verifier. Before sending HTML to the stock browser, the Glue downloads all `<cache>` tag requests. The HTML Verifier transforms the URLs of all tags that cause secondary interchanges to point to the Glue instead. In this way, the resulting transformed HTML can be sent to the stock web browser without fear of triggering inappropriate secondary interchanges.

5.6 Implementation

The HTML Verifier works by applying rules to an XML parse tree representation of the HTML document. It is easy to see how the behavior described above can be translated into a table-driven checker. The HTML Verifier must keep track of which nodes are in the public or private HTML segment. It then rejects certain nodes if they fall in a private segment.

For these rules to make sense every element of the tree must be identified as having been written entirely by the public or the private clilet segment. It is easy to determine the public and private segments of the HTML stream; after all, they are written by different API calls inside the public and private sandboxes, and are only later combined inside the Clilet browser. However, this does not guarantee that every element of the parse tree is either public or private. For that guarantee, the following initial constraints must be applied:

Atomic Tag Writing: Tags must be written entirely by the public or the private clilet segment, but not in part by both.

Private Sub-Tree: The HTML output of one call to the private clilet segment must consist of zero or more well-formed HTML sub-trees. It is not allowed, for example, for the private clilet segment to provide a matching end tag to a begin tag produced by the public clilet segment.

Well-formed XML: The clilet output must be a well-formed XML document. Although it may not be legal HTML, it must at least be parsable by a non-validating XML parser.

We now describe how we enforce these initial constraints in our prototype browser.

5.6.1 Atomic Tag Writing

We enforce atomic tag writing by providing a constrained API to the output stream that only allows tags to be written atomically. Rather than the standard stream output function such as *putchar()*, two functions are provided. The clilet author who finds these API calls to be cumbersome can build a standard stream interface to them:

***writeText()*:** Writes arbitrary text to the output stream; however, it converts any occurrences of the < or > characters into their HTML equivalents, ">" and "<". It is therefore impossible to write a tag to the output stream via the *writeText* call.

***writeTag()*:** Writes a begin or end tag to the output stream. The clilet provides the name of the tag and the value of all attributes to write; the *writeTag* then writes an entire, complete tag. Tag writing is therefore atomic.

5.6.2 Private Sub-Tree

In order for the HTML Verifier to tell the difference between public and private HTML segments, the Clilet browser adds extra tag information to the HTML output stream whenever a call is made to the private clilet segment. Just before a call, it adds `<private>`; just after, it adds `</private>`. The *writeTag()* API function does not permit either clilet segment to write these tags. Therefore, by keeping track of the `<private>` tags, the HTML Verifier knows definitively which HTML segments were written by which clilet segment.

In addition to marking context switches, the `<private>` tags enforce the private sub-tree property: by wrapping them around the output of the private clilet segment, the resulting document will be well-formed XML only if each private HTML segment consists of zero or more well-formed HTML sub-trees.

5.6.3 Well-Formed XML

The first thing the HTML Verifier does when receiving clilet output is to parse the output with a standard non-validating XML parser. Because of the `<private>` tags described above, this not only ensures the stream is well-formed XML, but also the private sub-tree property is enforced.

5.6.4 Verification Rules

Once the initial constraints have been satisfied, the HTML Verifier applies its table-based verification rules. These are the main rules preventing the private clilet segment from writing HTML in ways that could leak data to the server, and they have already been described in Section 5. The verification rules prevent anchors, forms and similar elements in private HTML segments. They allow for, and distinguish between, public and private form elements. They prevent secondary interchanges from private clilet segments, but allow for them in a limited way through the `<cache>` tag. Finally, they prevent context switches at inopportune moments.

5.6.5 HTML Validation

The last step of the HTML Verifier, although we described it first, is to ensure that the document conforms strictly to the Clilet HTML specification. This is necessary, especially in our prototype that uses an ill-defined stock web browser as its front-end, to ensure that the clilet does not try to invoke undocumented or otherwise illegal browser features.

In our prototype implementation, all comments are removed from the HTML in this step as well. Some stock browsers interpret JavaScript or other "extensions" placed in comments.

5.7 Conclusion

The HTML Verifier checks the HTML after it is written by the clilet and before it is displayed on the user's screen. It is responsible for preventing information flow from the private clilet segment to the server via HTML. To do so, it applies a set of validation rules to the HTML. Examples of such rules include the prohibition of anchors and secondary interchanges in private HTML segments. Finally, the HTML Verifier ensures that the output conforms to Clilet HTML, a subset of XHTML 1.1 designed to disable dangerous browser features.

6 Clilet Applications

We began with some motivating examples for the Clilet system. Now, having described the Clilet system and shown why it is secure, we proceed to describe more applications of this technology.

6.1 Demonstrations

Our canonical demo Clilet application, which has been implemented, is a checkbook register that stores check descriptions as private data.

Our next implemented demo application is one in which the clilets try to subvert user data privacy in a number of different ways. In each case, we show how our prototype Clilet browser does not let this happen.

6.2 E-mail Worms

E-mail worms, commonly called viruses, are currently a costly problem for e-mail clients that allow the easy execution of code received via e-mail. The obvious solution to this problem is to disallow executable e-mail attachments. Unfortunately, that solution would prohibit legitimate e-mail attachments as well as the malicious ones.

By applying the same type of multi-domain sandboxing controls used in the Clilet system to e-mail attachments, E-mail clients could run attachments, even attachments that are meant to modify private data, with more assurance against malicious code.

6.3 Available-Anywhere E-mail

POP and IMAP exemplify two general approaches to e-mail management today. In POP, a user downloads all e-mail to the client and periodically deletes it off the server; the user is responsible for storing and managing e-mail. In IMAP, e-mail remains on the server and is displayed to the user on-line.

IMAP presents a complete view of the user's e-mail but requires that the ISP stores all the e-mail. POP works well as long as the user works at a single main computer, but not when the user is travelling. At that point, access to downloaded mail is suddenly lost, as the user switches to an IMAP-type protocol during the trip. The POP user is forced to deal with two e-mail clients, one at home and one on the road.

A clilet-based e-mail system could give the POP user a more seamless experience. To make it work, the user must have a trusted machine that can store the "client-side" e-mail database. Clilet browsers will be directed to that machine for their private database. Messages would be downloaded from the mail server, as they are read, via the Clilet protocol; the distinction between displaying and downloading a message would be blurred.

7 Future Research

In addition to enabling new kinds of web applications the prototype Clilet system and implementation open up a host of new research projects. These directions for research fall into the broad categories of extensions and generalizations.

7.1 Extensions

Enriched API The current system allows for arbitrary client-side processing in the clilet, but only limited access to client-side resources. Future versions should provide clilets with a richer API for client-side tasks; client-side GUI access, for example. The trick is to ensure that user data privacy is still maintained, even as the set of capabilities allowed to clients is enlarged.

Limited Disclosure The current Clilet system builds an impenetrable wall between private data and the server. This is a good first step, but some applications required a controlled disclosure of some data to the server. For example, a web-based vendor might wish to store credit card numbers as private data rather than on the server; the browser would then need to transmit that private data, in a controlled fashion, to the server. The trick will be to make a mechanism for limited disclosure that is general enough to be useful for many applications, yet still secure.

Improved Multi-Domain Sandbox We do not yet know how how to build a secure multi-level system, such as our multi-domain sandbox, in a single Java virtual machine. At this time, it is not clear what changes (if any) to the Java system would be required. Existing research on Java-based multi-level systems includes a flow-verification system [7] and the *JKernel*/*JServer* [4].

7.2 Generalizations

Cryptographic Clilets For simplicity, we have avoided the use of cryptography when possible in our prototype. It would be interesting to re-visit the design junctures at which these decisions were made, and use more cryptographic options. The result would be another Clilet-type system, with properties similar but different from the system we do build.

Combined Data The Clilet system accomplishes the feat of allowing data from two distinct domains to be combined on a user's screen in a flexible manner without subverting the security of that data. It is a first application of the idea of bringing the program to the data, without ultimately letting the program know what the data is. Other non-web applications for this technique are inevitable.

Lightweight Multi-Level Computing Our Multi-Domain Sandbox uses the same ideas as those in multi-level operating systems, except that it simplifies them in two ways. It uses multi-level ideas outside the context of an operating system, and it prohibits concurrency. It seems that this simplified form of multi-level computing could certainly have many applications not envisioned by the designers of the classical multi-level systems.

Beyond Public-Private We have used only two sandboxes in our system: one public and one private. It is theoretically possible to build a multi-domain sandbox with many more domains, and with complex rules governing possible information flow between the domains. How these systems might be built and reasoned about, and what they might be useful for, is a matter for further research.

Multi-Level User Interface Through the use of HTML, the Clilet system allows a web application to build a user interface with two distinct types of user widgets, public and private. It should be possible to build multi-level user interfaces outside the scope of web applications — a multi-user GUI widget set, for example.

8 Conclusion

The web enables more interesting and complex on-line transactions every year. Clilets, which allow web applications to work with client-side data without the possibility of the disclosure of that data to the server, are another step in this evolution. They solve the problem of user data privacy, a problem that until now has not been addressed.

We have shown that the Clilet system is practical today by developing the necessary client and server components. Our prototype allows the use of a Clilet application via any standard web browser, thereby leveraging the installed base of web browsers. We expect that in the future, web browsers that speak the Clilet protocol natively will be developed.

We have built some sample Clilet applications, and we invite the interest to take a look:
http://www.eecs.harvard.edu/ citibob/clilet We also look forward to others interested in building and deploying new Clilet applications.

References

[1] D. Bell and L. LaPadula. *Secure Computer Systems: Unified Exposition and Multics Interpretation*. The Mitre Corp, 1976.

[2] Mary Campione and Kathy Walrath. *The Java Tutorial Second Edition: Object-Oriented Programming for the Internet*. Addison-Wesley Pub Co, 1998. `http://java.sun.com/docs/books/tutorial/index.html`.

[3] Dorothy E. Denning. "A Lattice Model of Secure Information Flow". *Communications of the ACM*, 19(5):236–243, 1976.

[4] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eiken. "Implementing Multiple Protection Domains in Java". In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, 1998.

[5] Carl E. Landwehr. "Formal Models for Computer Security". *ACM Computing Surveys*, 13(3):247–278, 1981.

[6] G. Lowe. "Defining Information Flow". Technical Report 1999/3, University of leicester, 1999.

[7] Andrew C. Myers. "JFlow: Practical Mostly-Static Information Flow Control". In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[8] Andrew C. Myers and Barbara Liskov. "Protecting Privacy Using the Decentralized Label Model". *Software Engineering and Methodology*, 9(4):410–442, 2000.

[9] Geoffrey Smith and Dennis Volpano. "Confinement Properties for Multi-Threaded Programs". In Michael Mislove Stephen Brookes, Achim Jung and Andre Scedrov, editors, *Electronic Notes in Theoretical Computer Science*, volume 20. Elsevier Science Publishers, 2000.

[10] G. Wagner. "Multi-Level Security in Multiagent Systems". In *1st International Workshop on Cooperative Information Agents (CIA-97)*. SV, 1997.