Großer Beleg

Plugin-based Isolation of Web Applications

Thomas Zimmermann

27th January 2008

Dresden University of Technology
Faculty of Computer Science
Institute for System Architecture
Chair for Operating Systems

Professor:	Prof. Dr. rer. nat. Hermann Härtig
Tutor:	Dipl.-Inf. Norman Feske

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 27th January 2008

Thomas Zimmermann

Abstract

A new approach to provide reliable security to web-based applications is presented. These web applications, i.e. websites that are generally referred to as Web 2.0., are composed from a set of web pages and scripts that are executed in a web browser. Unfortunately, today's web browsers no not provide effective security mechanisms. This work describes a mechanism that allows for the easy implementation of a given security policy, combined with high usability and portability between browsers. The web application, its input channels, and output channels are separated from the browser and installed on an isolated, trusted software stack on the user's computer. The application is then embedded into a web page by using the browser's plugin mechanism.

Acknowledgments

I would like to thank Norman Feske for mentoring this work. Originally, it was his idea to use plugins for building a security mechanism.

I also thank Neal Walfield. He was reading drafts of this document and gave me feedback and tips on its improvement.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Before and in the beginning of the World-Wide Web, most software applications were locally installed binaries. Their installation was commonly done by the system's administrator or a privileged user.

Some of these applications contained malware programs, i.e. programs that provided some non-advertised service with the intend to harm the user. This was not much of a problem at that time because computers were weakly connected and the installation of malware programs could be prevented easily by following some simple rules, e.g. using a virus scanner on all executables or compiling the binary from reviewed source code.

Applications running concurrently on the user's computer, were separated by the boundaries of virtual address spaces. Access to files or resources was restricted by some form of access control. This alone often lowered the damage done by malware programs to single user accounts or resources.

Over the last decade, the development of various techniques, made it possible to move applications towards the World-Wide Web. Such web-based applications are quite common today. There are interfaces for mailboxes or news services and even complete office suites and image manipulation software is available on the web.

Web-based applications consist of some web pages and a set of scripts and plugins which are loaded from an HTTP server and executed in a web browser.

While being some of the most flexible and often used application on today's computer systems, browsers do not provide any effective security mechanisms by default. All websites are loaded into the same address space. If not very special care is taken, for example the content of an online banking application could be accessible by another loaded website. Also, many users store their passwords in a local keyring. This is loaded by the browser and unlocked by the user entering his/her master password. Once the password is entered, the keyring's content could be exposed to any software, that has access to the browser's address space. Section 1.3 gives some ideas of how this could be achieved.

Another problem is the fact that all common user interfaces, e.g. X11 or the Windows desktop, are not designed with security in mind. The basic problem is that all running applications can read all input and output devices at any time they want. This makes it very easy for an attacker to spy for security-sensitive data, if he/she has access to the user interface.

Much work had been done to secure the channel between two computer systems, but in many cases the systems themselves are the weak spots. Matt Bishop gave a description of the problem [Bis03]:

> Cryptography provides a mechanism for [...] preventing unauthorized people from reading and altering messages on a network. [...] The canonical example is the use of cryptography to secure communications between two low-security systems. If only trusted users can access the two systems, cryptography protects messages in transit. But if untrusted users can access either system (through authorized accounts or, more likely, by breaking in), the cryptography is not sufficient to protect the messages. The attacker can read the messages at either endpoint.

Bruce Schneier has a similar opinion. In a discussion [Sch07] with Marcus Ranum he said:

> You [Marcus Ranum] are right about the endpoints not getting any better. I've written again and again how measure[s] like two-factor authentication aren't going to make electronic banking any more secure. The problem is if someone has stuck a Trojan on your computer, it doesn't matter how many ways you authenticate to the ban[n]ing server; the Trojan is going to perform illicit transactions after you authenticate.
>
> It's the same with a lot of our secure protocols. SSL, SSH, PGP and so on all assume the endpoints are secure, and the threat is in the communications system. But we know the real risks are the endpoints.

This work's intend is to improve the security at the user's endpoint. This is done by working out which components and system services need to be trusted and providing a framework that allows for programming of secure, web-based applications.

1.2 Definitions

Because this work is related towards security, this term needs to be further specified. I follow the definition given by Matt Bishop [Bis03]. Here security consists of

- confidentiality,
- integrity, and
- availability.

Confidentiality is the concealment of information or resources. This is done by providing some sort of access control. Only authorized users are allowed to read the confidential data. An example of such a mechanism is an encrypted disk drive where users have to enter a password when mounting the disk. Only authorized users should have this password. If an unauthorized user obtains the password then the confidentiality policy is compromised.

Integrity reflects the level of trustworthiness. This typically means that no unauthorized user is allowed to change data, especially not unnoticed. One can further distinguish between source integrity and data integrity. The former refers to the source of an information as being integer, i.e. to be the one it claims to be. The second refers to the data as being integer, e.g. not receiving a 0 if a 1 was send without noticing the change.

Availability describes the chance of getting a requested service. The higher this chance is, the higher is the availability.

Further, some security-related terms need to be defined. A *security policy* is a set of rules, formal or informal, that describe the security properties of a system. A system that reliably implements a security policy is called a *secure system*. Because it is hard to formally prove the correctness of systems, this reliability often comes from the user's believe in a system's security. A system that is believed to be secure is called a *trusted system*. The *Trusted Computing Base* (TCB) of a system is the minimal set of components in which the user has to trust. Its complexity is measured in *lines of code*.

1.3 Examples

This section gives some example attacks on web-based applications. The first two both are based on the design of the underlying user interface, i.e. X11, which does not provide isolation between its applications. The third example describes how an attacker could expose security-sensitive data by using the browser's plugin mechanism to get access to the browser's address space. A more formal problem description is then given in Section 1.4.

1.3.1 Logging User Input in X11

Input logging is a possible attack on the confidentiality policy of information entered by the user. Most users trust their user interface to deliver the entered information to exactly the application it is intended for. Unfortunately, this is not the case with today's common window systems.

Logging input in X11 is very easy. Each application connects to the X server and opens one or more windows on the screen. Input is then communicated from the X server to the application by sending event messages. In the X server's default setup, any application receives any input event. It is the application's task to filter out those events that are received by its own windows. In this design an application can also listen for events received by other application's windows.

The famous program *xeyes* is a good example of how this works. *Xeyes* tracks the pointer's position on the screen by listening for events of type *MotionNotify*. It then displays a pair of eyes on the screen where the eyeballs watch the pointer's current position. Figure 1.1 illustrates this.

If an attacker listens for events of type *KeyPress* instead of *MotionNotify*, he/she can log the user's complete keyboard input. This makes it very easy to spy for passwords or other confidential information.

The only solution X provides to overcome this problem, is to let an application grab an input device. This applications then receives the input from the device exclusively. Only one application can grab an input device at a time and any other application that wants to grab it, has to wait until the grabber releases it.

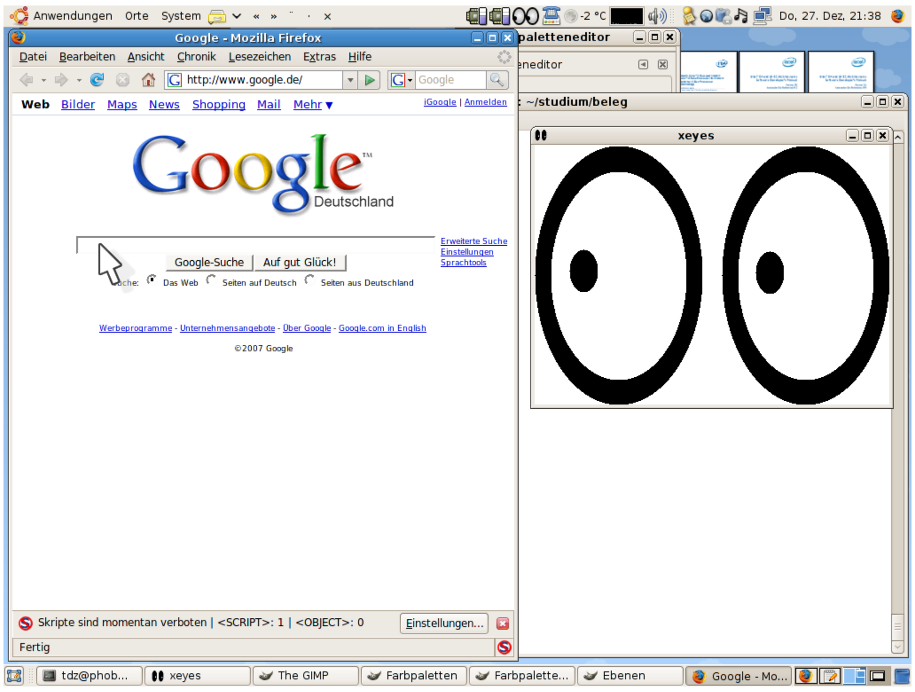

Figure 1.1: Xeyes, watching the web browser.

This mechanism can be used to separate single application's input from each other but does not provide sufficient isolation between components of the same application. Instead the mechanism's all-or-nothing strategy for grabbing is a good opportunity for a denial-of-service attack. Here an attacker grabs a device but never releases it.

1.3.2 Logging Output in X11

Output logging is another possible attack on a confidentiality policy of information. While input logging is a way to get confidential information entered by the user, output logging can be used to retrieve confidential information that is delivered to the user, e.g. data from online banking applications or the user's encrypted emails.

Output logging in an X environment is almost as simple as input logging. In a first step the spying application makes a screenshot of the screen. Again the default setup of the X server does not provide any mechanism to prevent this.

In a second step, the application applies a pattern recognition algorithm to the screenshot to filter out the information it is interested in. This is a complex problem. Text or image recognition software can be slow when applied to large data sets. A possible optimization is to reduce the screenshot to some subsection or to reduce the resolution.

1.3.3 Compromising Mozilla Plugins

This section gives a description of how to get access to the web browser's address space by using the browser's plugin mechanism. Further it describes an easy way of how to spy on other running plugin's content. Some technical details in this section refers to browsers of the Mozilla family, e.g. Netscape or Firefox, or some alternatives with compatible plugin interfaces.

An attacker can get access to a web browser's address space by writing a plugin which is then loaded by the browser. To get loaded by the browser, the plugin has to be installed on the end user's computer and a website has to request an instance of it.

The hardest part is likely for an attacker to get his/her malicious plugin library installed on the user's computer. This can be done by using a phishing website, where visitors are coerced to download the plugin's binary by making them believe they need this binary to use the provided service.

Once installed and loaded, the plugin can modify the browser settings to get loaded each time the browser starts. A simple idea for such a modification is to change the browser's default web page, which is loaded on each startup automatically. Here, the malicious plugin replaces the default page with one that starts the plugin. This way, the plugin code gets executed on each browser startup.

To easily expose the content of other running plugins, an attacker could attempt to break security by using the plugin mechanism itself. This can be done by using some sort of man-in-the-middle attack, as illustrated in Figure 1.2. Here the attacker's plugin sits between the browser and the actual plugin that implements the service, which is called *service plugin* from now one.

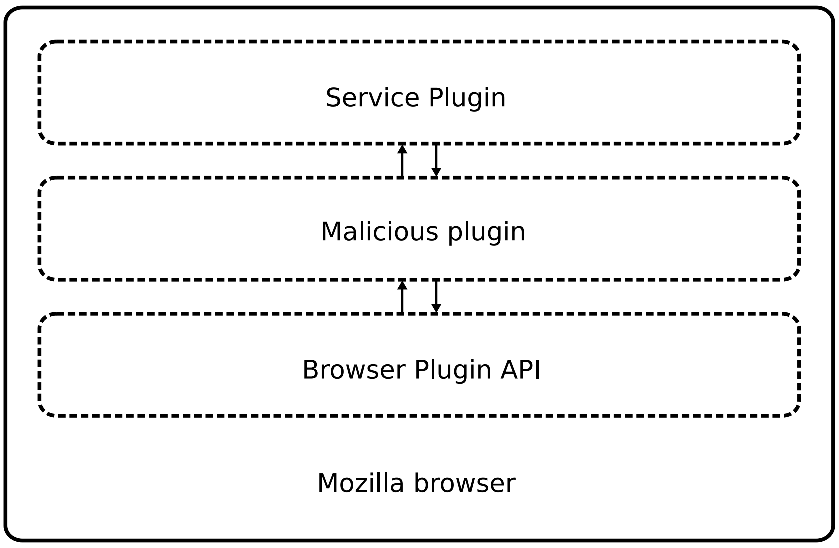

Figure 1.2: A malicious plugin, loaded between browser and service plugin, controls their communication.

Mozilla browsers have a set of standard rules, where to look for plugin libraries. The algorithm looks at the following locations:

1. A global directory, e.g. `/usr/lib/mozilla/plugins`,
2. The directories stored in the environment variable `MOZ_PLUGIN_PATH`, and

3. A place in the user's home directory, namely `$(HOME)/.mozilla/plugins`.

To get his/her plugin loaded between the browser and the service plugin, the attacker needs to make sure that it is found first, according to the search rules[1].

By copying itself into the user's home directory and cleverly modifying the content of `MOZ_PLUGIN_PATH`, the malicious plugin can control all other plugins, except those that are globally available.

The attacker's plugin then registers itself for the service plugin's MIME content type [FB96]. From now on the browser will start an instance of the attacker's plugin whenever a website requests an instance of the service plugin. This is a possible violation of an integrity policy.

An attacker can find the service plugin by using the same search heuristic as the browser. This way he/she can load the service plugin later on and let it provide the desired functionality. This makes it almost impossible to detect such a man-in-the-middle attack for any common end user.

Depending on the concrete intention of such an attack, it can compromise any of the three foundations of security. An attacker, who is interested in breaking the confidentiality of a secret stored by an instance of the service plugin, can log the content of this instance's memory.

Logging the content of dynamically allocated memory is simple. Browser and plugin communicate via a set of function pointers. These pointers are exchanged during the plugin's initialization. Two of these functions, `NPP_Alloc` and `NPP_Free`, are used for handling of dynamically allocated memory. The attacker sits between the browser and the service plugin's instance. During the initialization, he/she replaces the memory-function pointers with his/her own implementations and can now observe the instance's dynamically allocated memory.

Another attack can be used to examine the values on the program's stack. Here the attacker makes regular snapshots of the stack content and searches it for security sensitive data. An attacker does not need the observe the stack all the time, but only when the plugin's code gets executed. He/She knows when this happens because he/she controls the set of function pointers that connect the browser and the plugin instance.

Newer browser versions implement a more object-oriented approach, where interfaces are provided by service-manager classes. The setup is still done with the basic API, so an attacker can provide his/her own implementations of these classes.

When I prepared this work, I implemented a plugin to prove these statements. The plugin, once installed, searches for plugin libraries and registers itself for their content types. Whenever the browser opens a website that uses a service plugin, the browser loads my plugin. This, in turn, loads the actual service plugin's library which provides the requested service to the browser. Neither browser nor plugin notice that my plugin sits between them, acts like a man-in-the-middle and traces their API function calls, including arguments and return values.

[1]The order in which the locations are processed can vary between different browsers. Some, e.g. *Iceweasel 2*, first load the plugins in the user's local directory. This behavior allows for the control of global plugins as well.

The bottom line of this section is that a plugin implementing a secure service cannot trust its environment because the environment could be controlled by an attacker.

1.4 Problem and Solution

The given examples illustrate that web browsers do not provide adequate isolation between trusted and untrusted context' to make them suitable for working with security-sensitive data.

On one side, this is related to the security-weak environments that browsers are running in, e.g. the X Window System. These do not provide their applications with secure channels for input and output. An untrusted application can read any other application's input and output.

On the other side, the problem arises of the browser itself. The architecture of common web browsers allows for the loading of external code from untrusted sources. Such plugins are not well separated from each other and their environment but run in the browser's address space and therefore have the same access permissions as the browser does.

My approach to solve these problems is to isolate the security-sensitive components of web applications and put them in a distinct environment that is trusted to be secure with respect to a given security police. The web browser, running in an untrusted environment, interacts with the trusted web application via its Plugin API.

A website can use the application by instancing a web-browser plugin. I provide a software framework that does all setup and communication between the web browser and the trusted application. Special care is taken to ensure that the untrusted parts of the system can not compromise the trusted ones.

The trusted part of the system is based on an L4 microkernel. The L4 software stack provides software components that allow for the isolation of distinct security contexts from each other. Especially it provides a secure input and output channel between the application and the end user. The L4 stack has a small *Trusted Computing Base* of circa 300.000 lines of code. This makes it less error prone than the common web browser and its environment which has several million lines of code.

1.5 Overview

The rest of this document is organized as follows:

Chapter 2 introduces some previous attempts to solve the problems described above. I will summarize each approach and also examine its weaknesses.

The design for providing a secure environment to web applications is described in Chapter 3. Based on the weaknesses of the other approaches, first a set of design goals is derived. Then it is presented of how to realize these goals.

An implementation is given is Chapter 4. Is is based on an L4 microkernel environment and features a small *Trusted Computing Base* while allowing a very flexible usage.

Chapter 5 evaluates the design and implementation. It mainly discusses the improvements in security but also contains some information about the implementation's overhead.

Some ideas for future work on this topic are given in Chapter 6. One is to write a web application realized with the presented design. Another idea is to extent the current implementation and port is to other systems.

Chapter 7 concludes the work.

2 Related Work

There does not seem to be intensive research on reliable security in web applications yet. Many solutions in that field are build around virtual machines, such as Java or Java Script. These projects mostly try to solve a different problem. They isolate the application from its environment but not vice versa. For example, a program on the user's computer can receive the user's input into a Java application, even though the application runs in a virtual machine.

There are, however, a few research projects that focus on solving various aspects of the problem which is discussed here.

2.1 Flask on Top of XACE

A possible solution to the security problems described in Chapter 1 is presented in [Wal07]. The author uses the Flask Architecture's [SSL+99] security modules on top of the *X Access Control Extension* (XACE) [Wal06] to restrict access permissions of X clients.

The *Flask Architecture* is a mechanism for implementing fine-grained security in the operating system. The system provides a security server that implements a security policy and an object manager that enforces this policy. An client that wants to interact with the system, has to query the object server for permission. The object server itself queries the security server to decide about the client's query. Depending on the active security policy, the server either decides to authorize the query or not. If the query is authorized, the object manager delegates the permission to the client. Permissions can be revoked later on.

Flask was originally developed for the *Fluke microkernel system* [FHL+96], but it has been ported to Linux and is now implemented by SELinux [SVS06].

XACE is a generalization of former approaches to X11 security, e.g. SECURITY [Wig96] and Xtsol [sun]. In itself, it does not provide any security policies or take any decisions. It provides a set of hooks into the X server, such that third-party security extensions can interact with its internals. Whenever the X server gets a request from a client, it queries the registered security extension to decide about the request's validity. Only permitted requests are then executed by the X server.

XACE allows the security extension to examine all common types of events, such as core and extension-protocol requests, resource access, mapping and unmapping of windows, access to drawables and input devices.

By combining XACE and Flask, one can control the access permissions of X clients, based on a system-wide policy.

A typical scenario is an X server running on SELinux. It loads its Flask module on startup and registers its callbacks. On each client request, it calls the respective callback function, which than queries the SELinux' object manager. The object manager itself queries the security server to make the decision whether the X client's request should be permitted or not. The policy, implemented by this security server, is given by the system's administrator.

The combination of Flask and XACE is useful for securing X applications against each other but does not solve the security weaknesses of web-based applications because of the following disadvantages:

- The major disadvantage of this approach is that it only prevents simple attacks such as key logging or grabbing screenshots, i.e. attacks that use the security deficiencies of X.
- It does especially not provide any barrier between context' within the same application. If the attack comes from the inside of an application, i.e. another component in the same address space, it is still possible for the attacker to log an application's interaction with the X Window System.
- The approach does not prevent attacks on other low-level services or resource usage, such as denial-of-service or buffer overflows. For example if a loaded web page can somehow crash the browser or exploit a security hole, XACE does not help in any way.
- Additionally the Trusted Computing Base is large. It consists of a full Linux environment with hardware drivers, system services and an X server, which results in an estimated size of 10 to 20 million lines of code.

2.2 DarpaBrowser

The *DarpaBrowser* [SM02] is an approach to apply the *Rule Of Least Privilege* to the internals of a web browser. It is part of a larger solution called CapDesk which is based on Java and runs on common operating systems, such as Windows and Linux.

The idea behind DarpaBrowser is to separate all components of the web browser and the environment and assign individual capabilities to each component. For example, a component of a web browser that is only used for surfing the World-Wide Web does not need access to the user's local files. Another component that is only used for reading local files does not need access to the computer's network connection.

For every such action there exists a capability in the system. If a component is about to do an activity for which it does not own the necessary capability, the user is asked to assign permission or deny the request.

This approach is powerful enough to address many of the problems mentioned in the previous section, but it also has several disadvantages:

- With DarpaBrowser's design, security contradicts usability. Whenever a component demands special capabilities, the user has to decide whether the concerned action is allowed or not. Depending on the user's knowledge this decision may be too difficult. Although the project report specifies some general rules of thumb for such decisions, one cannot expect the user to know when a component needs a capability and when it does not.

- DarpaBrowser is not a drop-in replacement for today's common browser software. Its components were developed especially for this project and the user cannot replace some of them by versions of his favorite web browser. Given the enormous number of today's web technologies, this is a serious problem in terms of maintenance.
- The software runs in a non-trustworthy environment. Because of the weak foundation, provided by the underlying operating system, the browser is still exposed to attacks, e.g. third-party software, installed on the user's computer system, can easily log key strokes or grab screenshots.
- Again, the Trusted Computing Base is large. It consists of the operating system (i.e. Windows/Linux), a Java Environment, an interpreter for the programming language E and its own environment Edesk. This results in a TCB which I estimate to be of 25 to 50 million lines of code.

A more detailed review of DarpaBrowser can be found in [WT02]. My conclusion is that DarpaBrowser's approach is fine for securing the web browsers internals against each other, but does not provide sufficient protection against threats on the OS level.

2.3 Web Browsers in MashupOS

Another approach to build a secure environment for web applications is part of MashupOS [WFHJ07], a project by Microsoft Research. Instead of trying to solve the security problems on the level of the operating system, it focuses on improving the capabilities of the web browser and HTML.

MashupOS attempts to build a browser system that allows for multiple principals in the same browser process at the same time. It uses a modified version of the Internet Explorer. The modification mainly consists of an extension to the HTML parser and an implementation of a system of virtual machines that provide isolation of contexts and allow for the fine-grained setup of resources.

Two different types of virtual machines are supported by MashupOS. A *sandbox* can be used to isolate content from its containing web page, such that it cannot escape. In contrast, a *service instance* can be used by content to isolate itself from its surrounding container, such that the container cannot get in.

Both types of virtual machines are implemented by the web browser, which therefore needs to be trusted by either container and content.

To create an instance of a virtual machine, container or content use special HTML tags that describe the requested service. Each instance of such a virtual machine is provided with its own set of resources. These resources are memory, persistent state information, i.e. cookies, Document-Object-Model resources, and network connections. For each resource there exist rules of how the virtual machine's content can interact with this resource.

MashupOS is quite powerful with respect to the isolation of distinct security contexts within the browser but is also has several disadvantages:

- Content and container need to trust the web browser. This is not a good assumption, because if the browser or its environment is compromised, security-sensitive data might be exposed. An example of how to compromise the web browser is given in Section 1.3.
- The designers choose to use a modified version of HTML. Web pages that use this dialect of HTML are unlikely to display correctly on any browser that does not implement MashupOS. Because there are already many incompatible or half-standardized version of HTML and HTML parsers, website authors have to take special care to ensure portability of their content between different browsers. These circumstances are likely to hinder the adoption of another HTML dialect.
- Like the example of DarpaBrowser, MashupOS suffers from its security-weak environment and large Trusted Computing Base of estimated 25 to 50 million lines of code.

MashupOS is not suitable to solve the presented security problems of web-based applications, because all content and containers need to rely on the browser's trustworthiness.

Nevertheless, a browser-independent and standardized version, implemented on top of a more light-weight code base, can improve security in web applications considerably, while keeping it easy for authors to create web pages.

3 Design

Neither the web browser nor the underlying user interface is designed to provide security-sensitive web applications with an environment that does not allow for the abuse of an established security policy. Providing such an environment is the problem that this work is going to solve.

This chapter presents a new design approach to the security of browser-based applications. In Section 3.1, the goals of this design are introduced and in Section 3.2, it is shown of how to achieve these goals.

3.1 Design Goals

This design's major intention is to provide the benefits of the examples given in Chapter 2, while improving or fixing most of their drawbacks. Examining these examples, I see three major goals to achieve. These are

- improved enforcement of security policies, with respect to the definition given in Section 1.2,
- browser independence, and
- good usability.

The rest of this section discusses each of these goals.

3.1.1 Improved Security Policy Enforcement

Security has been defined in terms of

- confidentiality,
- integrity, and
- availability.

A system that reliably implements a security policy concerning these properties is called a *secure system* with respect to this policy.

Because improving security is the principal motivation behind this work, the *way of being secure* needs to be further examined. For implementing a *secure application*, i.e. an application that reliably implements a given security policy, one needs to provide at least three different security contexts that are designed to follow the given paradigms. These are

- the operating system,
- the application's environment, i.e. the web browser and system services, and
- the application itself.

The design and implementation of all these components needs to allow for the implementation of a security policy, e.g. a policy that requests the restriction of file access is not implementable on systems that do not have a notion of file access permissions.

Design and implementation should also not contradict the security policy. If a policy requests the authentication of a user by login, any system with a login mechanism can implement this policy. If there is, for some reason, an anonymous guest account with sufficient access permissions, the system contradicts the security policy, e.g. Windows 95 shows such a behavior by default.

This work only deals with the local environment of secure web applications. It does not deal with securing the application itself or the underlying operating system. From now on, it is assumed that each of these components in the software stack allows for the implementation of a given security policy and does not contradict it. It is also assumed that communication over a network is secure, e.g by using an encryption mechanism.

The basic idea of this work is to provide a secure environment to the web application by separating

- address spaces,
- input channels, and
- output channels,

of both, web browser and security-sensitive application. The application is executed in a trusted software environment, isolated from the rest of the system. It does almost all interactions within this environment and does not use any untrusted services. Communication between the browser and the application is reduced to a minimum. Section 3.2 discusses these concepts in detail.

3.1.2 Browser Independence

A *browser-independent solution* is a software component that can be used with most web browsers without the need for customization of the browser or the component itself.

This is crucial to make effective use of the software. Today's typical web browsers consist of several million lines of code. Even forking and adapting single components is an enormous task in terms of cost and maintenance.

Additionally, given the large number of network protocols, file types and web technologies that modern web browsers need to handle, it is illusory to build a specialized version of a browser while trying to keep up with the latest developments in that field.

Browser independence also means that one cannot change the syntax or semantics of the HTML document format. This would break compatibility with other vendor's browser software and web designers would hardly adopt such a technology.

Given the strict isolation between browser and web application, a browser-independent mechanism is needed to reconnect them for some minimal interaction that is needed.

In this work, I solved that problem by designing the connection mechanism on top of the plugin mechanism provided by modern web browsers. Hereby a website can use a trusted web-application by instancing a plugin which connects to the secure application in the trusted environment. While plugins are not fully browser independent, they are at least quite portable and do not require any modification to the browser's code base.

3.1.3 Good Usability

Usability refers to the ease of working with a software program. The more affordable it is for a user to achieve his/her intended goal, the more usable a software is.

Security sometimes contradicts usability. Experience [Yee03] shows that users typically will not use secure software solutions if they are too complicated to understand or use. Especially if the software requires some effort by the user to provide security, the user will often choose the insecure option. Remember the example of DarpaBrowser in Chapter 2. DarpaBrowser explicitly asks the end user to grant permission to a software component. He/She now needs to figure out if the requested permission is really required or not. This is not trivial in most cases. Even deciding by some rules of thumbs is more complicated for the user than just granting the permission without a second thought.

Another point is the good integration into the visual appearance of the system. The trusted component's interface should behave as the rest of the system does. Ideally the user should not even notice that he/she is working in a security-sensitive context. If the trusted application's interface behaves the way the untrusted browser's interface does, the user only has to learn one interface paradigm, instead of two. This will make the software more accessible and will increase its adoption by end users.

Having separated the browser and the security-sensitive web application, their visual interfaces need to be reunited to make them appear as being one application.

In this design, the browser uses a plugin to connect to the secure application. Such a plugin's instance has a visual representative that is part of the surrounding web page, i.e. a window of the native, non-trustworthy user interface. The application has its own, separated representative that provides a secure, trustworthy channel between itself and the end user.

The idea is to synchronize the visual representatives of plugin and application. The application's representative is always displayed in front of the plugin's representative. If the plugin's representative is moved, the application's representative is moved accordingly; if the plugin's representative is resized, the application's representative is resized accordingly. This gives the user the impression he/she is interacting with the plugin's representative, while in reality he/she is interacting with the application's security-sensitive representative.

3.2 Design Choices

The foundation of the design is to have the user installed two distinct software stacks on his computer.

One is the common setup, providing the user's applications, services, games, et cetera. On top of this stack runs the user's web browser. The components of this software stack are not trusted.

The web application, containing security-sensible data, runs on a separate software stack on the same machine, in parallel to the untrusted software. Each component on the stack, including the web application, is trusted by the user to confirm to the security policy. While it is often possible that common users or applications can modify the untrusted software stack to a certain amount, this behavior should not be possible within the trusted stack, i.e. a web browser could provide the option of automatically downloading untrusted add-ons, and install them in the user's home directory. It should however never be able to install a component in the trusted software stack. Ideally each component of the trusted stack is installed by a privileged user, who carefully reviews the component first.

Both, browser and application, are connected via a trusted channel, i.e. an IPC mechanism, provided by the operating system.

The design is illustrated in Figure 3.1. It isolates web browser and application from each other. To provide this in an effective way, one needs to separate the following items, as mentioned in Section 3.1.1:

- address spaces, i.e. the application's current state,
- output to the display device, and
- user input.

To increase the usability, the composed output of browser and web application should look to the user as if the application's output was a just another component of the loaded web page. This is done by providing a plugin which connects the web page to the web application and allows for the embedding of the application's output into the web page.

The rest of this section describes the separation of the browser's and web application's address spaces, the details of embedding an application's output channel into a web page and the input channel's properties.

3.2.1 Address-Space Separation

Address-space separation puts an effective barrier between the trusted application's data and the web browser. This provides confidentiality and integrity to the application's content. Because many errors do not propagate over address-space boundaries, it provides some sort of sandbox to the application, such that errors in the web browser cannot easily compromise the application's availability.

The separation of different processes address spaces is a standard feature in today's operating systems, which guarantees its availability on a wide range of platforms.

Plugin

The plugin is a module of the web browser. It runs in the browser's address space and is therefore suspect of being compromised by an attack. Thus, the plugin is not trusted.

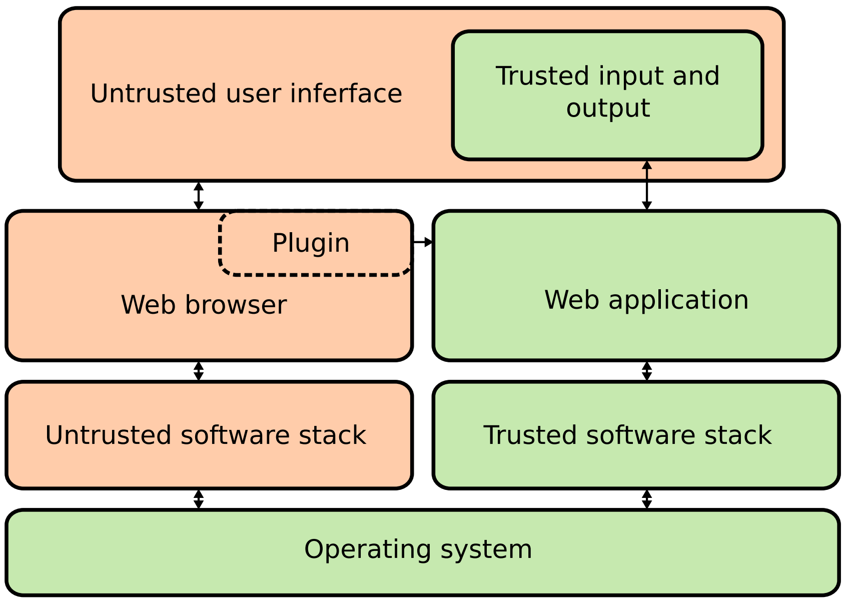

Figure 3.1: The design is based on a trusted and an untrusted software stack. The output of browser and application is combined by the user interface. Trusted components are colored green.

A plugin instance does not contain application data, especially no data that is sensitive with respect to security. Its only task is to act as a proxy between the web browser and the web application and control starting and stopping the server process.

Each plugin module is loaded by the web browser's built-in plugin loader. This will typically happen when a web page requests the use of a plugin. The tags `<embed>` and `<object>` are common HTML syntax for that case. Each plugin module is associated with one or more MIME content types [FB96]. The browser checks all plugin's content types against the type requested by the web page and selects the correct module.

When the plugin module has been loaded and the browser requests the creation of a new instance of the provided service, the plugin starts an application process from a trusted source and establishes a connection. The named HTML tags allow for the specification of parameters. These are passed as program options to the application.

Starting the application needs to be done in a way that does not allow the plugin instance to stop it afterward without the application's consent. For example, if the plugin instance is capable of stopping the application directly, an attacker, who has control over the plugin's address space, could do exactly this and provoke a denial of service.

The only way for the instance to stop the application should be sending a *quit* message and requesting the shutdown. This guarantees the application's availability, even if the plugin instance is compromised. The exact mechanism of such a design depends strongly on the underlying operating system. A very general idea is to let the trusted software stack provide a component, which starts applications for the plugin. This service has to be the only way for a plugin instance to start a new application. It has to make sure that no untrusted part of the system is able to stop the application afterward.

After the initial setup phase is finished, a plugin instance only acts as a mediator between the web browser and the web application.

Web application

The web application is a trusted component, running on top of a trusted software stack. It contains all algorithms and application data, especially the security-critical ones. An application is started by the plugin whenever the browser requests a new instance of the plugin. There is exactly one application per plugin instance.

Each web application provides its own input and output facilities to the user. These facilities are separated from the rest of the user interface and provide the user with a *trusted path* for interacting with the web application.

Very often an application needs to interact with other components in the system, e.g. connecting to a web server via network. Because the application implements a trusted service, it should interact with other trusted services as much as possible. The application therefore refers to its underlying trusted software stack. Imagine a web application implementing a service that provides online-banking facilities. The bank's web page starts the application on the user's computer and passes over some parameter to signal it of where to connect to the bank's server. The application then uses the network system of its underlying trusted software stack to connect to the bank[1].

IPC between Plugin Instance and Web Application

The plugin instance indicates to the application of how to adapt to changes in the browser's output. The application therefore listens on the IPC channel for incoming commands from its associated plugin instance.

Because of the weak connection between plugin instance and web application, there are only few commands to handle. These do

- the initial setup,
- window handling, and
- allowing the plugin instance to signal its shutdown.

An important point here is, that the application only reacts on commands coming from its associated plugin instance. For reasons of security, it may not accept commands from other sources. Else an attacker from the outside could send malicious commands to the application, e.g. provoke a denial of service by signaling the application to quit its service.

[1]Of course, additional encryption and authentication protocols need to be implemented when transferring data over the network. These are, however, outside the scope of this work.

In the initial setup phase the plugin instance is loaded by the user's web page. It starts the application process. The application then needs to call back to its plugin instance, to signal that it is waiting for commands. Depending an the underlying IPC channel, plugin instance and application often need to exchange some id value that make them addressable by each other, e.g. a process id or task number. Except for this initial setup, there should preferably not exist a channel from application to its plugin instance.

When the user unloads the web page, the plugin instance is about to be destroyed. It now signals to the application to quit its service. It therefore send the application a *Quit* command. As mentioned before, in a correct implementation there should not be a way for the plugin instance to kill the application process directly, to make the application's isolation more effective.

Information on window handling details can be found in the next subsection.

3.2.2 Output

The user typically interacts with a plugin's instance via the underlying graphical user interface. The web browser therefore provides each instance with a native window, where it can draw to and receive input from.

Because this window is known by the untrusted browser it is not trusted as well. Everything written to the window is supposed to be read by an attacker. The system needs to provide a secure channel from the web application to the output device.

The solution is to establish an overlay over the native window. This overlay is simply another window or drawing area that is displayed in front of the plugin-window's position. It provides a secure output channel by definition. It is up to the implementation to ensure that this assumption holds.

To provide features of availability, i.e. being visible on the screen, and integrity, i.e. not displaying compromised content, the overlay is only controlled by one trusted component, namely the application. Also the window system has to provide labeling information where each overlay is clearly identifiable by the end user. This sets up a *trusted path* between the user and the application. To guarantee confidentiality of the presented information, the window system may not allow for grabbing screenshots of this overlay.

If the overlay's visual appearance is designed similar to the surrounding web page, the whole system is completely transparent to the user. Because the overlay is perceived as being part of a web page, the user can easily associate each overlay with its corresponding web page. Because the overlay looks and behaves like the plugin instance's native window, the user does not need to change his/her habits when interacting with the secure context. On the other hand, it is still possible for the application programmer to indicate the secure context to the user by displaying some special visual appearance in the overlay, e.g a red border along the overlay's edges that mark the trusted channels boundaries.

The Overlay's State on the Screen

To keep the state of the overlay in sync with the plugin window, the plugin instance tracks the user's input and searches for certain events on its associated native window or one of its parent windows. These events are

- moving,
- resizing, and
- changing the visibility state.

The *window tracker* examines these events, coming from the native window system, and computes the overlay's new position and visibility. This new state is sent to the application which, in turn, adjusts the overlay's appearance.

The important point here is that the application has the option of ignoring such state messages. Otherwise, if an attacker gets control of the window tracker, he/she could remove the secure overlay and display a corrupt overlay instead. Preventing such an attack automatically is complicated. A simpler approach is to allow the user to control whether the application accepts the instance's commands or ignores them. This can be provided as a separated button in the trusted user interface.

3.2.3 Input

Similar to the output, there needs to be a secure channel between the user's input device and the application.

To guarantee some sort of confidentiality, this channel must not allow for tracking key strokes or mouse input by third-party applications. Additionally it must not allow third-parties applications to send events to the overlay. Such a feature would compromise integrity or availability of the channel.

The formerly established overlay is used as secure input channel. The user, interacting with the plugin window, e.g. clicking buttons or entering text, is in reality interacting with the overlay that is drawn in front of the plugin window. Because the overlay is separated from the rest of the window system, logging its input events is not possible anymore. Again, it is up to the actual implementation to provide this isolation.

4 Implementation

The implementation is based on *Fiasco*, an L4 microkernel, and some basic system services, e.g. a memory manager, a program loader, a network service, and a file provider. A detailed description of the this software stack (Figure 4.1) can be found in [HHF+05].

The L4 system is used because of the small size of its components. This work is implemented on top of the TUDOS environment created by the Dresden University of Technology. Depending on the configuration, the application's security-critical components can have a Trusted Computing Base of only around 300,000 lines of code.

On top of the L4 system runs *L4Linux*, a port of the Linux kernel to L4 operating systems [HHW98]. It provides the end-user's work environment. This typically consists of the well-known Unix utilities, an X server and the user's desktop environment.

Graphical output is provided by *Nitpicker* [FH05], a minimalistic graphical environment for the L4 operating system. The X server uses Nitpicker for its input and output and is treated as any other Nitpicker client. Section 4.2 contains details about Nitpicker. More information is found in [FH04].

A Mozilla web browser runs in the X environment as a normal X client.

The actual implementation, is build on top of the *Mozilla Plugin API*, which is described in Section 4.1, and Nitpicker. With this software setup, we can get both, a small Trusted Computing Base for the security-critical components and a decent desktop environment for the user to do his/her work. Figure 4.1 illustrates this.

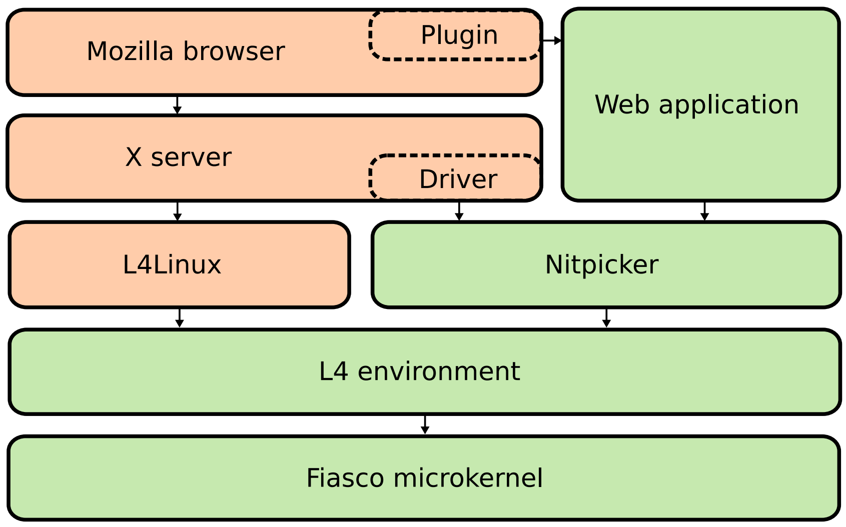

Figure 4.1: The software stack includes the Fiasco kernel, L4env, L4Linux, X, and Mozilla. Trusted components are colored green.

The complete software stack is based on standard components. No extra programming and very little customization is needed. All components are available under Open-Source licenses.

4.1 Mozilla Plugin API

The *Mozilla Plugin API* [Moz07] (NPAPI) is a set of functions built into Mozilla browsers to allow for the extension of websites by third-party plugins.

A plugin is a shared library, loaded by the browser at run time. Each library exports the symbol `NP_GetMimeDescription`. It is called by the browser and returns a set of MIME content types, handled by the library.

When a website requests a mime-typed object, the browser loads the plugin library into its address space and creates a new instance to start a session. At the end of a session the instance is destroyed and the plugin library gets unloaded by the browser.

Several distinct instances of each plugin can be in use at the same time. In the process of creating a new instance of the plugin, the browser and the plugin library exchange a set of function pointers that allow them to call each others functions.

The web browser's functions, supplied to the plugin, allow for getting or setting various browser properties and for allocating or freeing memory.

The plugin's functions, supplied to the browser, allow for getting or setting various plugin properties and allow the browser to assign a native window to the plugin, i.e an X11 window id. This window is used by the plugin to draw its output and receive input events from the underlying window system.

NPAPI was first introduced by Netscape 2.0 and has been in wide use for more then 10 years now. It has been ported to at least three different platforms and is supported not only by Mozilla-based web browsers, but also by Konqueror, Opera and their derivatives. This facilitates for providing browser independence of the presented design approach.

4.2 Nitpicker

Nitpicker [FH05] is a very simple window environment for the L4 operating system.

It provides the programmer with some simple abstractions of the graphic display. The application draws into *buffers* which are just continuous areas in main memory, describing RGB pixel data. Each buffer is made visible by establishing a *view* on the screen. A view displays the buffer on the visual output device. Each buffer and view is owned by exactly the application that created them.

Input is handled in a similar way. The user interacts with a view on the screen and the view's application receives an event message from Nitpicker.

Security has been the major design goal of Nitpicker. An application only knows and controls its own buffers and views. It can neither see the complete composed screen nor receive other application's input events. Additionally Nitpicker provides labeling facilities, such that end users know which view corresponds to which application, i.e a trusted path. This guarantees the properties of secure channels for input and output. Nitpicker's code complexity is lower than 2,000 lines of code.

Because of Nitpicker's superior security features, an overlay is implemented by a Nitpicker view.

4.3 Implementation Details

The implementation consists of two major components. On the untrusted side, there is a plugin that is loaded by the web browser and handles the connection of application and web page. On the trusted side, there is a framework that handles the command messages, coming from the plugin. Both sides are connected by L4 IPC mechanisms.

4.3.1 Plugin

The plugin is a shared library for the Mozilla browser. It is loaded by the browser at run time, based on the associated MIME content types.

A plugin is embedded into an HTML file with the tags `<embed>` or `<object>`. A MIME content type is supplied as parameter to these tags. When the web browser wants the loaded plugin library to create a new instance of itself, it calls the libraries symbol `NP_New` and passes the content-type string. The value of this string is used to start the correct web application on the trusted L4 software stack.

MIME-Content-Type Configuration

The plugin's MIME types are configured in a textual configuration file. The file contains the MIME type description for each MIME type to be handled by the client. A description consists of a MIME type, an associated L4 server program, i.e. the web application, and a descriptive string.

The configuration file is intended to be read-only for unprivileged users, so only a privileged user can install new plugins. Giving unprivileged users write permissions would allow attackers to install additional malicious services or remove trusted ones.

After being loaded into memory, the plugin is queried by the browser for a list of its MIME types. It reads the configuration file, composes, and returns an output string from this data[1]. This string is then parsed by the browser to determine the plugin's facilities.

Starting and Stopping an Application

Starting a web application needs to be implemented in the correct way. Otherwise, an attacker that controls the right components of the untrusted software stack can simply kill the application's task and therefore compromise its availability.

The application is a separate L4 task. Each L4 task has an owner task that is allowed to kill it. By default, this is the process that started the task. In this case, this would

[1]Some browsers cache these output strings between sessions until a new version of the shared library is installed. Adding or removing entries from the configuration file is therefore not necessarily detected. A simple solution is to *touch* [fsf] the plugin library's file when the configuration file has changed. The browser then detects the library's new time stamp and updates its caches.

be the plugin's process, i.e the web browser. An attacker, who controls the browser, can now simply kill the application's L4 task and provoke a denial of service.

The solution is to make the plugin instance fork a separate helper process which then starts the application task. After having done that, the helper returns the application's task id via a pipe and exits.

With this startup protocol, the application's owner is the helper process, which is already gone and can therefore not be compromised.

If the plugin instance wants its application to quit, it's only option is to send a *Quit* message, asking to application to exit by itself. The application now has the option to quit or stay alive.

Interaction with the X Window System

Each instance of the plugin receives an associated X window from the browser. This window is intended to be used as interface between plugin instance and user.

In this case, all security-sensitive input and output is done by the web application via Nitpicker. The application maintains a Nitpicker view, which is displayed at the position and size of the plugin instance's X window. This gives the end user the impression of working with the plugin window, while he/she is working with the Nitpicker view instead.

To keep the view in sync with the window, the plugin instance tracks the events coming from the X system and updates the view's geometry and visibility state (Figure 4.2). Ironically X' inability to provide any security features becomes very useful here.

For each window, the plugin instance forks a child process, dedicated to this task. This process is called *window tracker*.

The window tracker first builds a list containing the plugin window and its parent windows in ascending order. This is called *window stack*. It then registers for X events on these windows.

The window tracker now examines each incoming event and sorts out those, that describe the geometry or visibility of the window. Only a small number of events are interesting here. These are most notably the events typed as *MotionNotify*, *GravityNotify*, and *ConfigureNotify* for geometry changes and *MapNotify*, *UnmapNotify*, and *VisibilityNotify* for visibility changes. An event typed as *DestroyNotify* signal the window's destruction.

When processing a geometry event, the tracker walks through the whole window stack and computes the instance's window's global position and visible region. To compute the global position the tracker simply adds up all windows positions. To determine the visible region, the tracker takes the window stack's first entry and makes its geometry the currently visible region. It then walks up the stack by one and clips the currently visible region against the geometry of the current stack entry's window. The result of this clip becomes the new visible region. Successively doing this while walking up the complete stack results in the visible region of the plugin instance's window.

X distinguishes between various mapping and visibility states for its windows. These are *Mapped*, *Unmapped*, *Visible*, *PartlyVisible* and *Invisible*. Nitpicker views only know the states *IsVisible* and *IsNotVisible*. Therefore we can reduce the complex X states

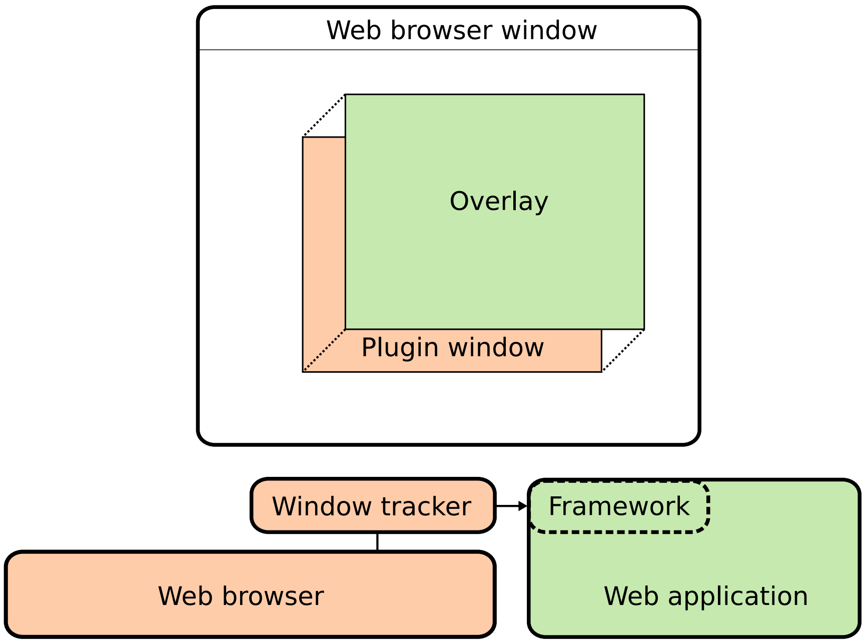

Figure 4.2: The overlay is kept in sync with the plugin's window by tracking X events and sending the overlay's new state to the application. Trusted components are colored green.

to these two cases. *Mapped*, *Visible*, and *PartlyVisible* are mapped to *IsVisible* while *Unmapped* and *Invisible* are mapped to *IsNotVisible*.

After successfully processing an event, the window tracker sends the view's new state to its application task, which updates the view's appearance on the screen. It then returns to the beginning of the event loop.

When the a plugin instance's window gets revoked by the browser, the client kills the associated window tracker. Because each instance of the plugin can only have one window at a time, there is never more than one additional process for each instance.

4.3.2 Web Application

The web application is implemented as an L4 task running in the trusted L4 environment. It links against a framework that handles the interaction with the client.

Each application is started and stopped by a plugin instance, such that each application is associated with one instance. Starting and stopping an application on demand has the benefit of not wasting the system's resources with unused, waiting applications. A non-existing task also cannot be target of an attack.

General overview

On startup, the application opens a connection to Nitpicker and creates an output buffer and view. It then initializes the implementation's framework with the Nitpicker connection, the view and a set of callback functions for incoming messages from the plugin. When the framework's init function returns, its main message handler runs as a service thread in the background.

The application continues its execution without the need to keep track of any plugin details. It renders its output to the Nitpicker buffers and receives input events via the Nitpicker view, it has supplied to the framework. All this is similar to a common Nitpicker application.

The framework's thread waits for IPC messages, coming from the plugin. These messages tell the application about the current position and visible region of the native window as well as its visibility state. When the window is visible, the server displays the supplied Nitpicker view as an overlay. This view displays its buffer's content according to the plugin window's properties.

Because the framework is completely generic and fully separated from the application code, each Nitpicker application can be turned into a web application with only a few lines of code.

Communication Setup

One problem for the plugin side is to find the correct framework thread in the first place. The application is started by the plugin instance via the L4 loader mechanism. This way, a instance only knows its application's task id, but not the id of the application's thread that handles the communication, i.e. the thread with the framework's message handler.

The solution is the implementation of a channel back to the plugin instance. Therefore, the plugin instance contains a thread which allows for the reception of the application's correct thread id.

The application is provided with the channel's server's thread id. This value is passed as program argument. It now calls this thread and sends it the complete thread id of the message handler.

After reception, the plugin instance knows where to send its command messages. The channel's thread is now stopped. It is not needed anymore and would only provide some interface for an attacker to interfere with the plugin.

Also, a plugin instance only waits for a given amount of time for the application to call, because if a website uses several instances at the same time, one needs to make sure that a failed startup of one of these instances does not propagate to the others. For example, if the first of two applications on a web page crashes while it is loaded, the plugin will continue with the second application after some time. It is then up to the web page to ensure that it can be operated correctly.

Selectively Reacting on Messages

Because the browser could be compromised by an attack, the application needs to provide the user with a way of restricting the plugin's influence, i.e. the end user has to be able to override the plugin's command messages.

To solve this problem, the framework provides some hooks into its message handlers. An application programmer using the framework for his/her plugins, can provide a callback functions for each message type.

If the framework receives a message, it first calls the registered function. The application then has the possibility to react on this message, e.g. increase the display buffer on a *Resize* message. Each callback function returns a Boolean value. If this is *True*, the current message is processed, otherwise the message is dropped. If the user is provided with a switch to disable the message handling, e.g. a separated button on the screen, he/she can decoupled an application completely from the plugin. Nitpicker's feature of labeling views gives the user a hint when the command messages might be compromised and decoupling is needed.

While security that relies on explicit user interaction is fragile [Yee03], this mechanism is still better than just letting the framework process possibly compromised commands.

5 Evaluation

An evaluation is done in terms of security and overhead.

5.1 Security

Security has been defined in terms of confidentiality, integrity and availability of information. Requirements on these qualities are defined by a security policy. A system is called secure if it reliably implements a given security policy.

The design, presented in this work, allows for the reliable implementation of a security policy. This section describes why this is the case and compares the presented implementation to the common Linux setup.

By separating the security-sensitive components from the rest of the system, improvements on all three qualities of security can be achieved. It is possible to implement a security policy with very strict requirements on confidentiality, integrity and availability of information.

Because of the strong isolation, it is much more complicated for an attacker to access security-sensitive information from input, output, or the application itself. An attacker can not

- read the application's input and output,
- get security-sensitive data by having access to the browser's address space, or
- simply provoke a denial of the application's service by crashing the web browser.

I tested these scenarios in a common browser setup[1] and on top of the presented implementation.

I used xeyes to track mouse events entered into an application. The common setup allowed xeyes to receive these events. With the secure setup, this was not possible. I did not explicitly test output logging because the presented implementation already relies on that functionality.

Getting access to the browser's address space was done as described in Section 1.3. I used a plugin to trace API calls between the web browser and some third-party service plugins. At the end of each test I had a complete log of all their communication, including passed arguments and return values. This makes it possible to easily read and write the security-sensitive memory content of the plugin instance in the common setup. In the secure setup this is also possible, but the plugin here only acts as a proxy between the

[1]My system is a Debian *Etch* Linux with Xorg 7.1 and Firefox 2.0.11.

browser and the application. It does not contain any security-sensitive data. This makes the attack useless.

A final test was about the robustness of the design. I crashed the browser process to see if that could possibly harm the application's availability. This was the case in the common setup. If the browser crashed, all loaded web pages are not available anymore. In the secure setup, this was not the case. A crash of the browser led to the unavailability of the loaded web pages, but the application's overlay was still visible and the application was working as expected.

The results of my tests come from the strong isolation of browser and application. There are still cases where both connect with each other. These are rare and only occur to make the application's output behave similar to a common plugin, e.g. change its position or size when the user manipulates the surrounding window. The user always has the option of disabling this behavior, such that the application is fully separated from the browser.

In general, the presented design provides an increase in security over the common setup. Attacks on the browser or its environment, as described in Section 1.3, do not work anymore.

If an attacker wants to break security, he/she now has to attack the trusted software stack. The security of this stack relies on the quality of the trusted components that are used. If one of these components is broken, any security mechanism might be useless.

It is hardly possible to measure software quality in absolute values. Therefore, *quality* is here defined as the number of errors in a software. The less the number of errors is, the higher is the quality.

The number of errors can be estimated. Therefore, the minimal number of trusted lines of code[2] between the presented implementation and a common software stack on Linux is compared. A conclusion is then drawn from statistical data.

Counting the number of lines of code is a very simple method of analysis. When doing this there are some details to be kept in mind. It is assumed that similar packages use similar programming language. It is quite meaningless to compare different implementations of the same algorithm if these implementations use very different languages, especially if one of these languages is specialized for algorithm's problem domain[3]. Most of the components in the software stack are implemented in C or C++, so this assumption holds.

Further it is assumed that the programmers of similar components have a similar level of experience. It seems rather unlikely that an expert on programming has as many programming errors in his code as a novice programmer, even if the novice programmer has a higher knowledge of the problem domain.

As mentioned before, the number of code lines in the Trusted Computing Base for the classic plugin design is quite large. On a Linux system it is more than 10 million lines. Table 5.1 shows measurements for some common components.

[2]The numbers of lines of code where generated using David A. Wheeler's 'SLOCCount'.

[3]One could argue that the use of a specialized language results in less lines of code and therefore less errors in the implementation. While I see the point, I have not been able to find any reliable data on that topic, so I do not assume this here.

Component	Lines of Code
Linux kernel 2.6.23.14	5,497,571
GNU C library 2.7	980,574
Xorg 7.3	566,328
Gtk+ 2.10	513,090
Mozilla Firefox 2.0.0.11	2,783,739
$\sum$	10,341,302

Table 5.1: These are the lines of code of some typical components in the Linux software stack. Each of them contributes to a common plugin's Trusted Computing Base.

A common assumption is that there is *one error per 1000 lines of code.* For the given setup, this gives a total number of errors somewhere near 10,000 errors. Even if there is no intention to attack this system, an error might be exposed by chance, which can result in a loss of security.

Component	Lines of Code
Fiasco kernel	97,138
UC C library	199,190
System services (roottask, names, dm_phys, loader, bmodfs)	16,729
Nitpicker	1,815
Framework library	2,952
$\sum$	317,824

Table 5.2: These are the lines of code of the major components in the L4 environment. The Trusted Computing Base of a plugin is substantially smaller than on a Linux-based system.

In an L4 environment the application's Trusted Computing Base is significantly lower. As shown in Table 5.2 it is somewhere near 300,000 lines of code. Using the same *one-error-per-1000-lines* assumption as above, the estimated number of errors is 30 times smaller than before, near 300 errors.

My conclusion is that a web application, running on the isolated L4 software stack, is less error prone and therefore very likely more secure than the same implementation running as conventional plugin library on a common Linux system.

The concrete numbers of lines of code can be doubted. Depending on the configuration and compile-time options, the source code that is finally compiled into binaries can vary. Some components, e.g. the Linux kernel or Xorg, come with various device drivers included, where the user only runs a small number of them. But even though the user might only run 10 percent of the Linux stack's code, the point of my argumentation remains true.

5.2 Overhead

Another issue is the speed of operation. The presented implementation is based on a microkernel system, which are commonly suspected of being slow in their execution.

The implementation's overhead depends mainly on two factors: the overhead of the underlying IPC operations and the performance of the secure overlay's display refresh. Both issues are discussed in this section, but in rather general terms. Also, I have not measured any of them explicitly. I found this pointless, because this work focuses on security, not optimization.

All communication in this microkernel system is based on the L4 IPC mechanism. This IPC was designed with the overhead of context switching in mind. It is therefore faster then low-level communication in many other operating systems.

With the framework, IPC mostly occurs when the plugin's window is moved or resized. These operations are implemented as *Short IPC*, where values are passed from sender to receiver via processor registers. No expensive memory copying is involved here.

The test machine is an AMD Sempron 2200+. I did some real-world tests, i.e. moving the window, switching between programs and dragging other windows around. All these tests result in IPC operations that notify the application that some property of the overlay has changed.

In fact, I have not been able to notice any serious delay while working. There is a small delay when the plugin is started but this is related to hard-disk IO and mostly negligible.

The tests and results for the overlay's display output are similar. Nitpicker provides a display driver based on standard VESA modes. It is not highly optimized but provides sufficient speed for working.

Testing the display's speed is done with some OpenGL software rendering. The test is based on a Nitpicker port of *Mesa3D* [mes], a free implementation of OpenGL.

A version of the famous program *glxgears*, shown in Figure 5.1, displays rotating gears in a Nitpicker view. Successively porting it to the framework implementation allowed me to run it similar to a web application.

The demo's default frame rate is 20 Hz. I have been able to run four instances at the same time without much noticeable delays in display refresh.

I do not think that display speed is a limiting factor. There is also a lot of optimization possible. Nitpicker's display driver is not optimized for speed and the port of OpenGL uses a rather inefficient form of double buffering that includes coping between the display buffers. If modern rendering techniques where applied to this components, there should be no speed difference to implementations on Linux or Windows.

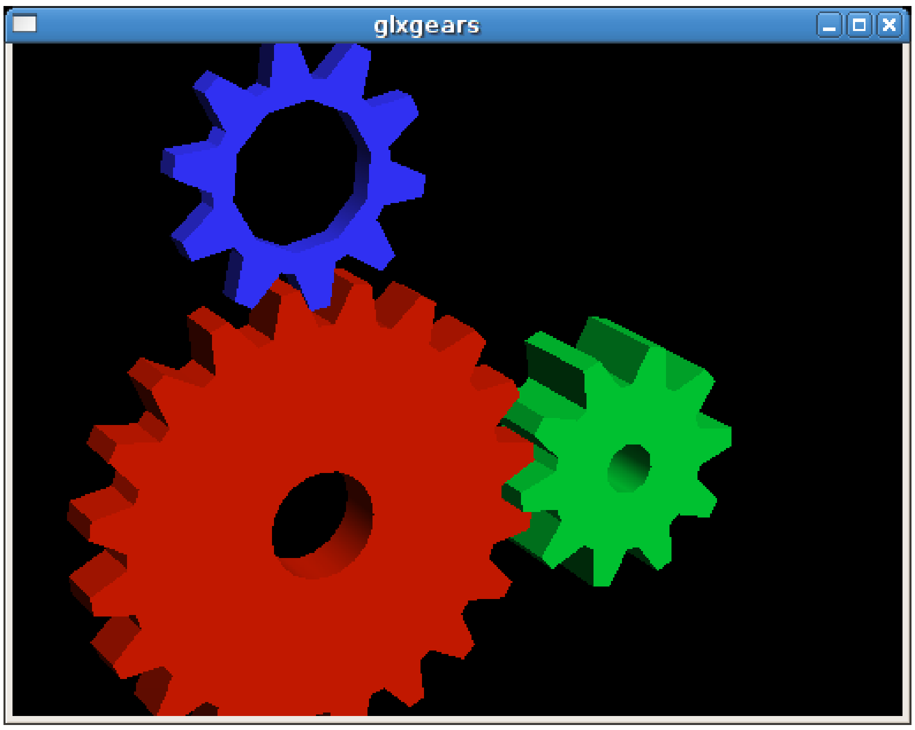

Figure 5.1: The *glxgears* demo displays 3 rotating gears.

6 Future Work

The first and foremost task is the implementation of a real application on top of the framework. An idea is to port Java to the trusted software stack of this setup, such that programmers can implement Java applets which automatically use the framework internally. Imaginable use cases include online-banking websites, ticket-booking services, online shops, or services in a company's Intranet.

With some work it should be possible to use common, third-party plugins with this design. The idea is to add a program to the trusted software stack that loads the plugin and provides it with an NPAPI interface. This program than acts as the trusted web application. A minimal X environment has to be added as well. The common plugin now runs on top of this X and uses its Xlib API. The X outputs the plugin's display to the Nitpicker view. Such a setup can load any plugin, while providing the presented security properties.

The implementation can still be extended. At the moment, it only provides Mozilla's basic plugin interface. Some more work has to be done to provide scripting or even Java support on the plugin side. On the other hand, I do not expect that the use of these features makes much sense here. The current, simple interface has its benefits in security.

Another task is to port the software to Linux- and X-based systems. By now users have to install the underlying L4 system. Some people might prefer a pure Linux implementation, because Linux is much more common and might provide some additional features that L4 is missing, e.g. specialized network services.

Each web application would be a separate Linux process in the system. It had to be forked by the browser itself or some external loader service[1]. Securing the input and output channels is more complicated. An idea is to use the XACE extension of newer X servers for achieving some access control over X clients. An evaluation has to be done if this works out.

The Linux version would lack the feature of a small Trusted Computing Base with all its benefits. On the other hand, having a widely-distributed system that provides greatly improved security at the user's end point is surely better then today's situation with almost no security at all.

[1]Having this as the default behavior is not only wishful in terms of security but would also result in a much more reliable browser. An error-prone plugin could not easily crash the browser anymore.

7 Conclusion

This work presented a new approach for implementing secure web applications. It is almost completely browser independent by using the plugin mechanism, provided by modern web browsers.

The design is based on separating the application's (1) address space, (2) output, and (3) input from the rest of the system. The isolated application is provided with a trusted channel to the input and output devices. It is executed on top of a trusted software stack. The application's visual appearance is integrated with its surrounding web page via the browser's plugin mechanism.

This separation results in an increased (1) confidentiality, (2) integrity, and (3) availability of the web application's data and service.

Together with the design, an implementation is provided, that uses (1) L4 services, (2) L4Linux, and (3) Mozilla as its basic components. The user employs his/her Linux software in the L4Linux environment. If the web browser supports Mozilla's Plugin API, any security critical application can now be implemented in the L4 environment. This fulfills all established design goals and has a Trusted Computing Base of only $\tilde{3}00{,}000$ lines of code.

Bibliography

[Bis03] Bishop, Matt: *Computer Security.* Addison-Wesley, 2003. 1, 2

[FB96] Freed, N. and N. Borenstein: *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*, November 1996. `http://www.ietf.org/rfc/rfc2045.txt`, visited on 19th October 2007. 6, 17

[FH04] Feske, Norman and Christian Helmuth: *Overlay Window Management: User interaction with multiple security domains.* Technical Report TUD-FI04-02, Technical University Dresden, March 2004. `http://os.inf.tu-dresden.de/papers_ps/ovl-techreport.pdf`, visited on 21st December 2007. 21

[FH05] Feske, Norman and Christian Helmuth: *A Nitpicker's guide to a minimal-complexity secure GUI.*, December 2005. `http://os.inf.tu-dresden.de/papers_ps/dach2005.pdf`, visited on 19th October 2007. 21, 22

[FHL$^+$96] Ford, Bryan, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson: *Mikrokernels Meet Recursive Virtual Machines.* In *OSDI96*, pages 205–212, October 1996. `http://cs.utah.edu/flux/papers/fluke-rvm.ps.gz`, visited on 23rd November 2007. 9

[fsf] *Coreutils - GNU core utilities.* `http://www.gnu.org/software/coreutils/`, visited on 21st December 2007. 23

[HHF$^+$05] Härtig, Hermann, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter: *The Nizza Secure-System Architecture.* In *Proceedings of The First International Conference on Collaborative Computing: Networking, Applications and Worksharing.* IEEE Computer Society, December 2005. 21

[HHW98] Härtig, Hermann, Michael Hohmuth, and Jean Wolter: *Taming Linux.* In *Parallel and Real-Time Systems (PART'98)*, 1998. 21

[mes] *The Mesa 3D Graphics Library.* `http://www.mesa3d.org`, visited on 27th December 2007. 32

[Moz07] Mozilla.org: *Gecko Plugin API Reference*, August 2007. `http://developer.mozilla.org/en/docs/Gecko_Plugin_API_Reference`, visited on 19th October 2007. 22

[Sch07] Schneier, Bruce: *Schneier on Security: Security in Ten Years*, December 2007. `http://www.schneier.com/blog/archives/2007/12/security_in_ten.html`, visited on 14th December 2007. 2

[SM02] Stiegler, Mark D. and Mark Miller: *A Capability Based Client: The DarpaBrowser.* Technical Report BAA-00-06-SNK, Combex Inc, June 2002. `http://www.combex.com/papers/darpa-report/index.html`, visited on 25th October 2007. 10

[SSL$^+$99] Spencer, Ray, Stephen Smalley, Peter Loscocco, Mike Hibler, David Anderson, and Jay Lepreau: *The Flask Security Architecture: System Support for Diverse Security Policies.* In *Proceedings of The Eighth USENIX Security Symposium*, pages 123–139, August 1999. `http://www.cs.utah.edu/flux/papers/flask-usenixsec99.pdf`, visited on 25th October 2007. 9

[sun] *Solaris Trusted Extension.* `http://www.opensolaris.org/os/community/security/projects/tx/`, visited on 9th January 2008. 9

[SVS06] Smalley, Stephen, Chris Vance, and Wayne Salomon: *Implementing SELinux as a Linux Security Module.*, February 2006. `http://www.nsa.gov/selinux/papers/module.pdf`, visited on 23rd November 2007. 9

[Wal06] Walsh, Eamon F.: *X Access Control Extension Specification*, October 2006. `http://people.freedesktop.org/~ewalsh/xace.pdf`, visited on 19th October 2007, Draft only. 9

[Wal07] Walsh, Eamon F.: *Application of the Flask Architecture to the X Window System Server*, 2007. `http://www.nsa.gov/selinux/papers/xorg07-paper.pdf`, visited on 19th October 2007. 9

[WFHJ07] Wang, Helen J., Xiaofeng Fan, Jon Howell, and Collin Jackson: *Protection and Communication Abstractions for Web Browsers in MashupOS*, October 2007. `http://research.microsoft.com/~helenw/papers/sosp07MashupOS.pdf`, visited on 7th November 2007. 11

[Wig96] Wiggins, David P.: *Security Extension Specification, Version 7.1*, November 1996. `http://www.xfree86.org/4.7.0/security.pdf`, visited on 19th October 2007. 9

[WT02] Wagner, David and Dean Tribble: *A Security Analysis of the Combex DarpaBrowser Architecure*, March 2002. `http://www.combex.com/papers/darpa-review/security-review.pdf`, visited on 25th October 2007. 11

[Yee03] Yee, Ka-Ping: *User Interaction Design for Secure Systems.*, December 2003. `http://people.ischool.berkeley.edi/~ping/sid/uidss.pdf`, visited on 15th January 2008. 15, 27

Index