Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2017

André Ribeiro Lopes

Interfaces seguras para uma infraestrutura de exploração de serviços pessoa-pessoa
Secure interfaces for a person-to-person service exploitation Infrastructure

Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2017

André Ribeiro Lopes

Interfaces seguras para uma infraestrutura de exploração de serviços pessoa-pessoa
Secure interfaces for a person-to-person service exploitation Infrastructure

"Arguing that you don't care about the right to privacy because you have nothing to hide is no different than saying you don't care about free speech because you have nothing to say. "

— Edward Snowden

Universidade de Aveiro Departamento de
Electrónica, Telecomunicações e Informática,
2017

André Ribeiro Lopes

Interfaces seguras para uma infraestrutura de exploração de serviços pessoa-pessoa
Secure interfaces for a person-to-person service exploitation Infrastructure

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requesitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica de André Ventura da Cruz Marnoto Zúquete, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e de Fábio José Reis Luís Marques, Professor Adjunto da Escola Superior de Tecnologia e Gestão de Águeda

o júri / the jury

presidente / president	**Professor Doutor João Paulo Silva Barraca** Professor Auxiliar da Universidade de Aveiro
vogais / examiners committee	**Professor Doutor José Carlos Coelho Martins da Fonseca** Professor Adjunto do Instituto Politécnico da Guarda (Arguente)
	Professor Doutor Fábio José Reis Luís Marques Professor Adjunto da Universidade de Aveiro (Orientador)

agradecimentos / acknowledgements

Special thanks to everyone who supported me during these five years, my family, friends and everyone that I met along the way who contributed in some way to my evolution as a person and as a computer engineer. My acquired interest for cybersecurity and privacy is thanks to some people that crossed my way during my time as a student, without them this dissertation would almost certainly not be a reality.

Resumo

Com o surgimento de aplicações Web interativas a Web mudou para sempre. Web sites passaram a ser mais que as páginas estáticas, sem vida dos tempos da Web 1.0. Graças à linguagem Javascript estas páginas tornaram-se aplicações interativas capazes de competir com os seus equivalentes em desktop. Faz sentido, portanto, numa arquitetura orientada a privacidade para partilha de serviços que estes sejam aplicações Web, naturalmente usando Javascript. No entanto esta linguagem apresenta alguns sérios riscos de segurança, e como esta arquitetura proposta deverá tolerar a inclusão de aplicações Web third-party, permitir o uso de Javascript sem restrições é impraticável. Como remover todo o Javascript não é aceitável quando se pretende construir aplicações Web modernas e interativas, deverá ser usado, de alguma forma, Javascript controlado.

Muitas soluções desenvolvidas no passado tentam isolar Javascript confiável e informação sensível numa página Web com Javascript não confiável. A nossa abordagem é diferente no sentido em que não há uma análise do código Javascript na aplicação Web em si, como outras soluções fizeram. Em vez disso, é feita uma filtragem completa de todo o Javascript usado nas nossas aplicações e usamos anotações especiais em páginas Web para injetar o nosso próprio Javascript seguro. Assim, desenhámos e desenvolvemos um sistema capaz de efetuar parsing de certas palavras-chave (ou anotações) que se traduzem em código Javascript seguro, que permite o controlo completo de todo o código Javascript a correr numa aplicação acoplada nesta arquitetura.

Para acessar o número mínimo de anotações necessárias de forma a construir uma aplicação Web apelativa e reativa, desenvolvemos três aplicações diferentes que as usam: uma galeria de fotos, um chat por texto e um chat por voz.

Abstract

With the dawn of interactive Web Applications the face of the Web was changed forever. Web sites are no longer the lifeless, static pages of the past, thanks to Javascript they became full-fledged interactive applications, now competing with their desktop counterparts. Thus, for a privacy-oriented architecture proposed for service sharing, it was decided to design them as Web applications, naturally using Javascript. However, this language presents some serious security issues, and since the architecture should tolerate the inclusion of untrusted third-party Web applications, allowing an unconstrained use of Javascript in those applications is unacceptable. Since removing all Javascript is not an option when it comes to build interactive, modern, Web applications, we need somehow to use controlled Javascript.

Many solutions were developed in the past that try to isolate trusted Javascript and sensitive information on a Web page with unstrusted Javascript. Our approach is different because we do not analyse existing Javascript code deployed within Web Applications, as others did. Instead, we completely filter out any Javascript used by our applications, and we use special annotations in Web pages to inject our own, risk-free Javascript code. Thus, we designed and developed a system able to parse certain keywords (or annotations) that translate into safe Javascript code, which allows us to have a complete control over the Javascript code running on an application plugged into this architecture.

For asserting a minimum set of annotations necessary to build an appealing and reactive Web application, we developed three different applications that use them: a photo sharing gallery, a text chat and a voice chat.

Contents

List of Figures

Chapter 1

Introduction

In the beginning of the Internet, Web pages were lifeless static pages meant to present information, and not full-fledged applications like today. Information mainly flowed in only one direction: from the server to the client. The Web mainly functioned through requests from the browser on the client to the server, the server then returned an entire HTML Web page. Any change to the current HTML page being shown on the browser window would require a completely new Web page to be sent from the server with the updated information.

In order to provide an interactive interface for Web applications and sites, active client-side content started being supported back in 1995, together with Flash and Java applets. However, Javascript is today the most widely used to this effect; Flash is not supported in some systems (notably, Android) and Java applets are deprecated.

The introduction of Javascript as a technology of the World Wide Web (WWW), alongside with HTML (HyperText Markup Language) and CSS (Cascading Style Sheets), provided Web pages with a much bigger degree of interaction, allowing the development of sophisticated applications that made browsers quasi-Operating Systems. In fact, browsers became increasingly complex pieces of software, and a typical computer user spends most of its time using them.

The modern Web encourages users to use the browser and Web applications for almost anything, from communication using text, voice and video and file storage to the use of numerous Web applications, such as online office suits, which were previously used only as traditional desktop applications. Web applications are increasingly serious competitors to their desktop counterparts since they are extremely practical: there is no need to install software on our hard-drive, software is always up-to-date and all of our work and personal settings are stored online, being available on any device. This is very appealing and practical on an age where the access to the internet is being increasingly done through several personal mobile devices[1].

[1]http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide - accessed on 6/07/2017

1.1 Motivation

The research described in this paper was motivated by a new service sharing infrastructure. The goal of this infrastructure is to enable people to interact between each other, in a private, person-to-person fashion, instead of using a central service for each kind of interaction (e.g. Twitter for instant messaging, Skype for voice calls, etc.). The applications ran by participants can be produced by anyone (thus, they are not trustworthy, nor they came from trustworthy sources). Such applications, hereafter referred simply as P2PApps, are fundamentally service-oriented, peer-to-peer Web applications, and each person (user) interacts with their Web application (i.e. with the one they own) with browser.

In our trust model we do not want to trust on the P2PApps to behave correctly, thus we need to prevent P2PApps from misbehaving, i.e. from stealing personal information or destroying personal information accessible from where they run. In other words, we need to provide a sandbox, i.e., a secure confinement solution, for preventing malicious P2PApps from engaging in forbidden actions. But, for that, we do not want to trust on security features deployed by browsers, because those are fallible. In fact, an estimated 18% of Chrome users, 34% of FireFox users, 42% of Edge users and 51% Safari users still use an outdated version of their browser [3] which make them vulnerable to several attacks that circumvent browser built-in security. In 2015, Symantec reported 876 of these vulnerabilities on the most used browsers [4]. Finally, In the recent years browsers became capable of supporting third-party plug-ins that extend the original browser's features. Those plug-ins are frequently the target of many security exploits, Symantec reported 679 vulnerabilities on third-party browser plug-ins, Adobe plugins being the biggest culprit mostly because of Adobe Flash Player [4].

1.2 Problem

As previously referred, HTML, Javascript and CSS are nowadays the fundamental building blocks for developing an appealing and effective Web application interface. Thus, they are the natural choice for developing the Web interface of the P2PApps. Unfortunately, Javascript introduces a number of security issues that need to be tackled. In fact, Javascript is extremely powerful to manipulate the browser DOM (Document Object Model), thus it is fundamental to have a reasonably good degree of trust in Web sites providing Javascript code. Javascript has some specific problems due to the way this language is built, some of these issues can be avoided using defensive programming conventions [5]:

- All Javascript code is executed within the same global environment using the same engine and, as such, it is not possible to isolate objects from each other.

- Any Javascript object is freely mutated by any other object with access to it. Even though this language specifies the constraints *Internal*, *ReadOnly*, *DontEnum*, and *DontDelete* in Javascript object's properties, there is no way to express those constraints in the language.

Browsers' built-in security mechanisms often protect users from the most nefarious Javascript-based attacks. The same-origin policy is one of the main security mechanisms designed to prevent malicious Web sites from reading confidential information from another Web site. Same-origin policy restricts the access of a certain script or document to a Web page or resource from a different origin [6]. If two pages have the same hostname, port number and protocol, then they have the same origin [7]. This mechanism is the first line of defense when it comes to preventing cookie stealing and similar attacks. However, since much of the Web relies on shared resources between different Web sites, this policy may be relaxed in some conditions.

Concluding, our problem is the following: how can we allow P2PApps to use Javascript while, at the same time, preventing Javascript-based abuses independently of the mechanisms deployed or not by a Web browser?

1.3 Contribution

This dissertation describes a solution that we adopted to deploy secure Web interfaces for our P2PApps. Such security is independent from browsers' protection features and it relies on the total removal of all sources of problems in the Web contents delivered by P2PApps to browsers. This means removing from those contents all the elements that could trigger (i) unwanted access to the DOM and (ii) unwanted accesses to Internet locations other than the correct P2PApp.

The solution that we adopted was to reject all the Javascript code provided by P2PApps, as well as non-local references to resources (i.e., URI's using hostnames). However, since Javascript is critical for having reacting and appealing interfaces, we allow P2PApps to request the usage of safe Javacript libraries, which are provided by the sandboxing environment that encapsulates P2PApps. Those libraries are not directly invoked by Javascript code of P2PApps (because it is totally removed); instead, they are indirectly called by annotating the HTML contents with special tags. Those tags are processed by a mediator that stands between the browser and the P2PApp, which replaces them by the apropriate call to the Javascript libraries. This mediator is also the architectural component that sanitizes the interface contents provided by P2PApps, by removing from them all Javascript and non-local URI's.

For a proof of concept we developed 3 P2PApps for assessing their feasability: a photo gallery, a message chat and a voice chat. For deploying those P2PApps we also build a minimum runtime environment for them, allowing them to communicate with each other, emulating a future environment interconnecting two people.

Chapter 2

Javascript-based attacks

With the increased dependency on Web Applications, the motivation for ill-intentioned people to find and exploit weaknesses on those increases as well. Those attacks can often be done with very simple tools, sometimes a Web browser is just the tool it is needed. On top of that, the development of Web applications has increased greatly in the past few years, which led to the increase of developers, and not all of them are aware to the Web security-related issues that should be addressed from the start [8].

Javascript is a very flexible language that can be used to manipulate the DOM of a Web site and pull sensitive information such as cookies, user location, browser history and even track user's behavior through mouse movements, mouse clicks, mouse scrolling and clipboard content [9].

In the following sections we will briefly describe some attacks that can be performed with the help of Javascript code running on the victims' Web browser.

2.1 Cross-Site Scripting (XSS)

A Web site vulnerable to an XSS attack would put their users at risk of getting cookies or other sensitive information stolen, and possibly used by an attacker to impersonate the victim. The way this attack works is by tricking the victim to click a carefully crafted link with malicious code, or a link to a Web site that contains the malicious code [10], taking advantage of the confidence a user has in that site.

As an example, suppose that a Web site is vulnerable to cross-site scripting attacks. The site can be accessed by the following address:

```
http://regularwebsite.com/hello/name?=John
```

which would return a Web page with the HTML presented next:

```
<HTML>
    <Title> Hello John</Title>
</HTML>
```

An attacker could supply the victim with the following link:

```
http://regularwebsite.com/hello?name
                                =<script>alert(document.cookie)</script>
```

This would return:

```
<HTML>
    <Title> Hello<script>alert(document.cookie)</script></Title>
</HTML>
```

From an attacker's perspective this is not very useful, since the alert window would pop-up on the victim's browser and the attacker would not acquire the information displayed on it. A more interesting, and perhaps lucrative, alternative would be to send this information to an attacker-owned Web site as follows:

```
http://regularwebsite.com/hello/name?
        =<script>window.open("http://www.evilsite.com/collect.cgi?
cookie="document.cookie)</script>
```

The HTML which would be returned:

```
<HTML>
    <Title> Hello<script>window.open("http://www.evilsite.com/
collect.cgi?cookie="document.cookie)</script></Title>
</HTML>
```

The code inside the `<script>` tag would run on the victim's browser, prompting it to access the attacker's Web site sending the cookie stored for `regularwebsite.com`.

2.2 Cross-Site Request Forgery (CSRF)

This attack exploits a service to which the user is currently authenticated by performing unrequested actions on behalf of the user. In a sense, this type of attack explores the trust a Web site has in the user's browser. Consider, for example, a banking Web site called `verysafebank.com`, which uses this API for money transfers:

```
http://verysafebank.com/transfer?to=x&ammount=y
```

A malicious programmer named John Doe could develop a service containing the following piece of code:

```
document.write("<img
src='http://verysafebank.com/transfer?to=johndoe&ammount=100'/>");
```

A user loading the service currently logged on `verysafebank.com` would execute the code above and make, unknowingly, a money transfer to John Doe.

2.3 Drive-by downloads

Despite security mechanisms employed by browsers, frequently these are the target of attacks that aim to exploit specific security fragilities. Since, as explained on Chapter 1, many users are negligent when it comes to updating their browsers, because of this, these security weaknesses are kept on their browsers for far longer than expected.

Drive-by download attacks are downloads that occur without the knowledge or consent of a user. In Figure 2.1 we can observe the lifecycle of a drive-by download. If a user accesses a malicious Web site, a Javascript code snippet, exploiting the user's browsers or plug-ins, downloads malware which infects the user's machine. Then, the infected machine may be used to supply a botnet or other nefarious purposes, including installing a rootkit.

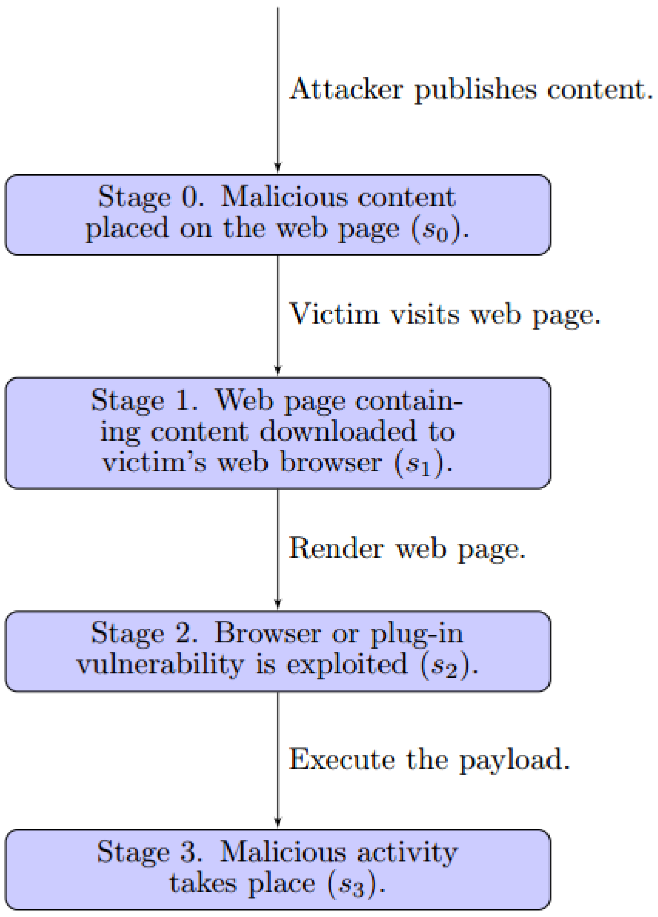

Attacker publishes content.
Stage 0. Malicious content placed on the web page (s_0).
Victim visits web page.
Stage 1. Web page containing content downloaded to victim's web browser (s_1).
Render web page.
Stage 2. Browser or plug-in vulnerability is exploited (s_2).
Execute the payload.
Stage 3. Malicious activity takes place (s_3).

Figure 2.1: Life cycle of a drive-by download attack [11]

To perform this attack either one of two situations must happen: A malicious service is written by an attacker or a legitimate one is compromised.

The attacker's code usually targets specific operating systems, browsers or plug-ins, looking for a known vulnerability that can be exploited. The exploitation of a known vulnerability usually leads to the injection of a payload on the victim's system containing code that will execute malicious activities. This can include stealing information or connecting to the Internet and download malware from an attacker's owned Web site [11].

2.4 Other

There are many other ways to develop a Web page using Javascript that can be seen as less serious but will, at least, degrade the experience on the end-user. Javascript may be used to develop difficult-to-avoid pop-up advertising. More seriously, using Javascript it is possible to tamper with browser configurations [12] and write a malicious script that hijacks the browser's saved favorites and change them, possibly to a link crafted to perform an XSS attack. Other types of attacks may include having flashing colors on the screen that might be troublesome for users prone to seizures.

Chapter 3

State of The Art

With the inherent security problems of Javascript [12] many attempts were done in the past to mitigate some of those problems. Particularly, this became a huge necessity with the rise of paid advertisement services, as Javascript ads became a majority of online advertisement, with many of these ads being written by third-party, untrusted organizations or developers.

Browser's plug-ins or extensions also attempt to deal with this problem, once being typically connected to a Web page DOM which may contain harmful Javascript code.

3.1 Blockstack

The decentralization of traditionally centralized networks or services is a great area of interest with the increased awareness of the privacy issues related to centralized architectures.

The work in this area aims to provide a safe, private environment to the development and use of Web applications.

Blockstack is a concept for a new decentralized Internet that aims to remove the traditional problem of blind trust on centralized services [13]. Blockstack's applications run locally using a secure domain name system instead of running on a remote server, making the user independent from third parties. The Blockstack is composed by three components:

1. Blockchain – Used to bind digital data to a public key, this way a new node on the network can independently assert all data bindings.

2. Decentralized Storage - Stores data values which are signed using a owner key. The user does not need to trust on the storage as they can verify its integrity independently.

3. Peer Network – Uses zone files, similar to DNS, to store routing information. Data storage is independent from discovering resources on the network, this way it is possible to integrate various storage services, both cloud storage services and P2P.

This concept for a decentralized Internet pushes the logic and complexity away from centralized servers to the end-user devices. Blockstack re-uses existing infrastructure to store encrypted data. The main idea is for the storage providers, such as Dropbox, to not have visibility into the user's data.

This way, Blockstack implements a decentralized person-to-person infrastructure using decentralized web applications with an interface written in HTML and Javascript.

3.2 Preventing Web based attacks

As discussed earlier, attacks to Web clients are a very common occurrence on the Internet. These attacks often result in theft of information or malfunction of these Web applications. Disruption on Web applications can often directly lead to the malfunction of a business or public service. With this in mind, preventing these kinds of attacks has increased relevance with the increased dependency of the society on the Web. Many solutions were put forward in the past to mitigate or completely eliminate Javascript threats or general Web-based attacks. Nevertheless, no silver bullet has yet been found, thus research on this area is still active.

3.2.1 Chrome Extensions

Google Chrome allows the inclusion of third-party software to enhance and customize the user experience in the form of extensions. Since Chrome is a browser with security as a priority on its design [14] Google developed a new extension platform with security in mind. A Chrome extension comprises multiple components, including content scripts, an extension core, and optional native binary. To mitigate and prevent extension vulnerabilities, Chrome employs the following mechanisms:

- *Privilege separation* – Chrome extensions adhere to a privilege-separated architecture. Extensions are made of two isolated components: content scripts and core extensions [15]. The core extension is ran on a separate process from the browser process, it contains most of the privileges, however it can only communicate with content scripts and any interaction with a Web page is done via `XMLHttpRequest`. This component does not have direct access to the host machine. A content script is written in JavaScript, it has direct access to the DOM of a Web page, and, therefore, may be exposed to malicious content. Because of this, the content script has the least privileges so it cannot access any object out of its renderer process space. All communication with the core extension is done via Chrome's interprocess communication (IPC) [1]. This separation of components can clearly be seen on the architecture diagram shown on Figure 3.1.

- *Isolated worlds* – While content scripts have access to the DOM of a Web page, the page's program heap is different from the content script's program heap. This way, a Web page cannot access variables or functions within the content script.

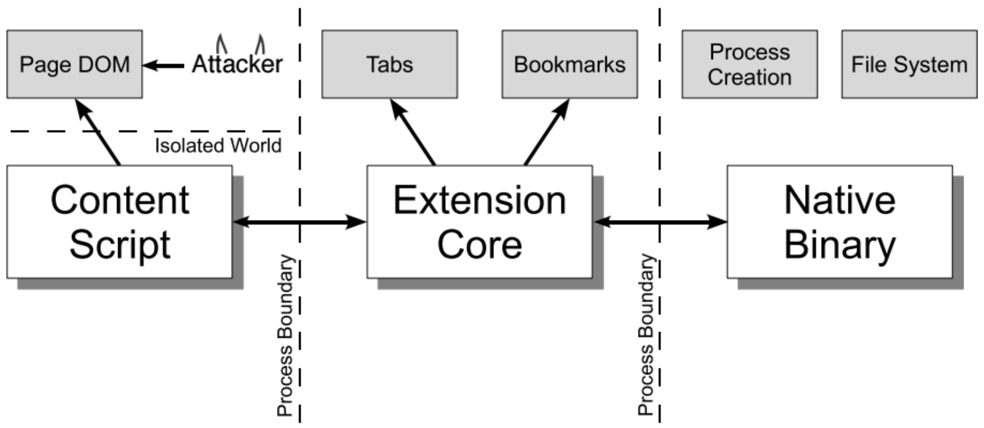


Figure 3.1: General architecture of a Chrome extension [1] containing its main components: the content script that has direct access to the DOM, the core which has restricted access to the DOM and the optional binaries that can interact with the operating system.

- *Permissions* – An extension is supplied with a list of permissions that restrict access to the browser's API and Web domains. In case a core extension is compromised, the attacker only gets the privileges supplied to the extension beforehand.

Optionally, an extension can have binary code [14]. These executables have direct access to the host machine, and, as such, are not protected by the security mechanisms discussed earlier and, therefore, represent a potential danger. However, their usage is very sparse and subject to previous manual review before being submitted to the Chrome Web Store [15].

3.2.2 Google Caja

Caja (from the spanish word for box, also short for **ca**pabilities **ja**vascript), is a tool to safely embed third-party HTML and Javascript code onto a trusted Web page. It is a subset of Javascript designed to make as little impact as possible on regular Javascript programming [5]. However, Caja programming differs from regular Javascript in a few ways:

- *Forbidden names*: Caja does not allow names using double underscore, this is reserved for internal use by Caja itself.

- *Frozen Objects*: A frozen object is immutable, its properties cannot be changed in any way, an attempt to do so will result on an exception being thrown. Prototypes and functions are frozen by default, this is the way to deal with Javascript's common attacks that involve rewriting functions at runtime.

- *No shared global environment*: Each module is isolated, preventing any abusive and dangerous access of each other's properties.

- *Internal names*: Property names ending on a single underscore are protected instance variables that can only be used with `this`. Like in Java or C++, protected variables are only accessible within the same chain of inheritance.

- *No method stealing*: Caja defines functions as simple, which do not need further safety measures, constructors (named functions that mention this) or methods (anonymous functions that mention this). Constructors or methods cannot be stored in variables or passed down as arguments.

- `eval` *and* `with` *removed*: Caja comes bundled with a JSON library that allows the most common uses of `eval`, avoiding all the possible risks of dangerous commands being passed to `eval`.

When embedding third-party untrusted software, like a game, on a trusted Web page using Caja, each game is embedded on a `<div>` and interaction between the game and the trusted Web page is done using regular Javascript objects (see Figure 3.2). Caja also supports the enforcement of certain policies, such as limiting how many notifications a game sends.

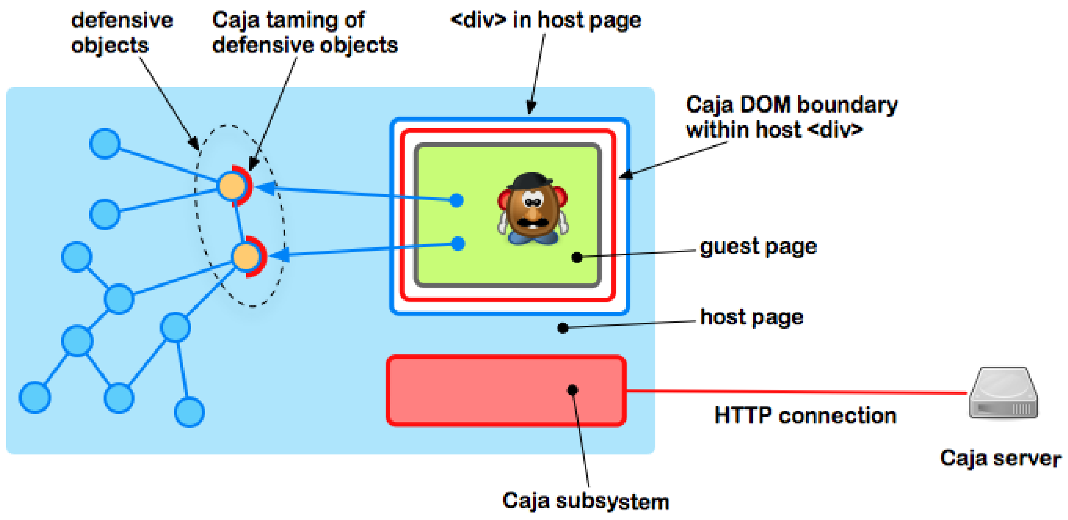


Figure 3.2: The Caja System.[1]

On Caja the following vocabulary is frequently used:

- `Host Page`: The container for the third party untrusted content.

[1]http://developers.google.com/caja/docs/about/ - accessed on 29/01/2017

- `Host code`: Code running on the Host Page.
- `Guest code`: Third-Party untrusted code that is embedded on the Host Page.
- `Guest Page`: If the Guest Code is a Web page containing code then it is called Guest Page.
- `Policy`: Developer decisions about what should or should not be allowed to be done by the guest code.
- `Defensive Object`: An object in the host page that is constructed with caution to provide only a limited authority to its clients. The host page grants limited authority to guest code by supplying it with the appropriate defensive objects.
- `Taming`: Current JavaScript objects are not able to make themselves tamper-proof. Taming is the process of registering defensive objects with Caja as provided them to guest code. Caja ensures that only the published API of objects is available to guest code and guest code cannot modify the object in ways not intended.
- `Cajoling`: The processing of making guest code safe to execute on the host page. This is done by including inline checks that make sure that invariants supplied to Caja are not broken and that the guest code does not refer to variables that it has no permission to use.

In Figure 3.3 the main flow of the Caja system is presented. To use Caja, first a `<div>` must be prepared to receive guest code(1). Caja must be included on the host page. The

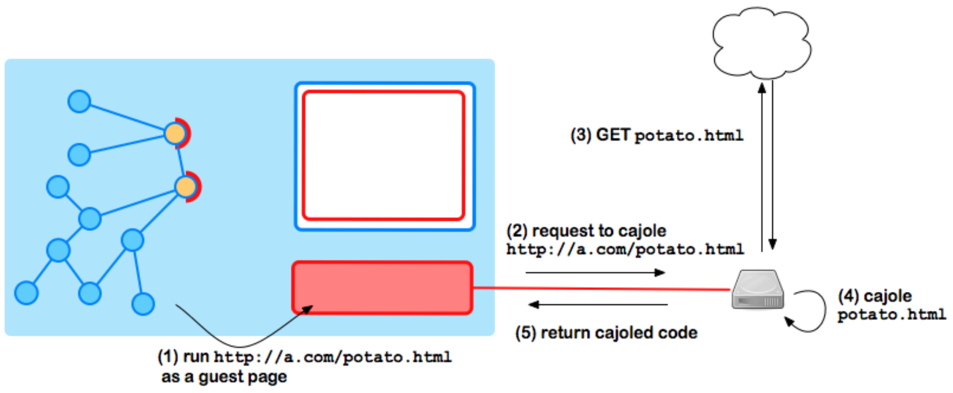

(3) GET potato.html
(2) request to cajole http://a.com/potato.html
(4) cajole potato.html
(5) return cajoled code
(1) run http://a.com/potato.html as a guest page

Figure 3.3: Main flow of the Caja system [2]

Caja script offers a connection to the Caja server, which can be the one that is publicly available or a custom one deployed by the developer.

[2]http://developers.google.com/caja/docs/about/ - accessed on 29/01/2017

The host code asks Caja to tame the defensive objects, and uses Caja to construct a DOM boundary within the `<div>` it as chosen (2).The host page then asks Caja to run the guest code within the `<div>` supplied. To make this code safe to execute it has to be cajoled (4) first, which is done via an HTTP GET to the Caja server defined previously (3), this GET returns the transformed code, deemed safe to run (5). Caja then runs this transformed safe code with the defensive objects defined earlier.

3.2.3 Facebook Javascript (FBJS)

The Facebook platform allows the development of third-party apps and games embedded within the Facebook enviroment [16]. Since this is a social network Web site, some sensitive information may be found on a personal page. It would be deeply problematic if third-party code could get this information by having direct access to the DOM. It was with this motivation in mind that Facebook Javascript (FBJS) was developed.

FBJS is a subset of the Javascript language. Facebook applications are written in FBML, a subset of HTML. FBJS is served from Facebook, after filtering and rewriting. To ensure that third-party code (or applications) has no direct access to the DOM, Facebook libraries mediate access to it. This mediation is done via adding application-specific prefixes to all top-level identifiers in the code, isolating the effective namespace of an application from the namespace of the host page, in this case, a page that is part of a personal Facebook profile. For example, the statement `document.domain` may be rewritten to `a12345_document.domain`, where a12345 is the application's specific prefix. Adding this prefix will prevent guest code from directly access the main document content. To access it, the guest application must use the provided Facebook libraries that mediate and control the access and interaction to the main document [17].

3.2.4 Yahoo!ADSafe

Third-party advertising providers are now common on the Web. While on the past advertising was just a static image loaded on a Web page, nowadays an ad ranges from a simple image to mini-games embedded on a Web page. Due to the nature of the advertising, ads aim to be as appealing as possible to the public. This often means, in the realm of electronic advertising, increasingly complex pieces of software. This high level of interaction provided by this advertising paradigm is powered by the heavy use of languages like Javascript.

Having content written in Javascript dynamically pulled from remote servers comes at the cost of having serious security risks since, if not properly isolated, a malicious ad could have access to the surrounding Web page, which could harbor sensitive information.

ADSafe is a subset of Javascript that allows the inclusion of advertising code directly on the host page. Interaction between unstrusted code and the host page is very limited, mediated through an ADSafe object, provided as a library that offers an API to safely interact with the DOM of the host page. As an example, the following code [17]:

```
var location = document.location
```

would be rewritten by the developer as:

```
var location = ADSAFE.get(document,"location")
```

3.2.5 BrowserShield

BrowserShield [2] is a javascript library that allows vulnerability-driven static and dynamic HTML inspection and cleansing, rewriting it to a safe equivalent using a set of policies, see the architecture diagram on 3.4. Being vulnerability-driven, BrowserShield transforms pages according to known vulnerabilities. This system does not alter the browser itself, so the logic injector could be used on a number of different contexts like client or edge firewalls, browser extensions, or third-party content providers, such as advertising companies. To safely transform any input into a safe equivalent, two separate translations are used along with policies enforced at run time in order to filtering any known vulnerabilities.

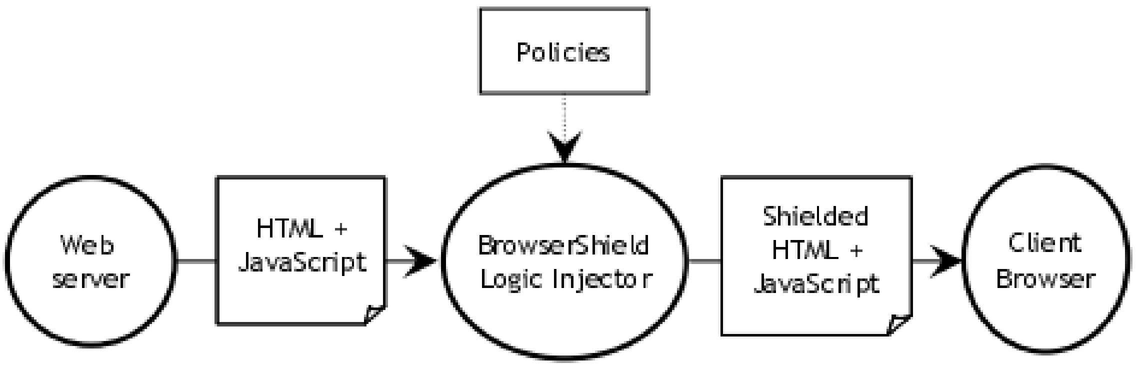
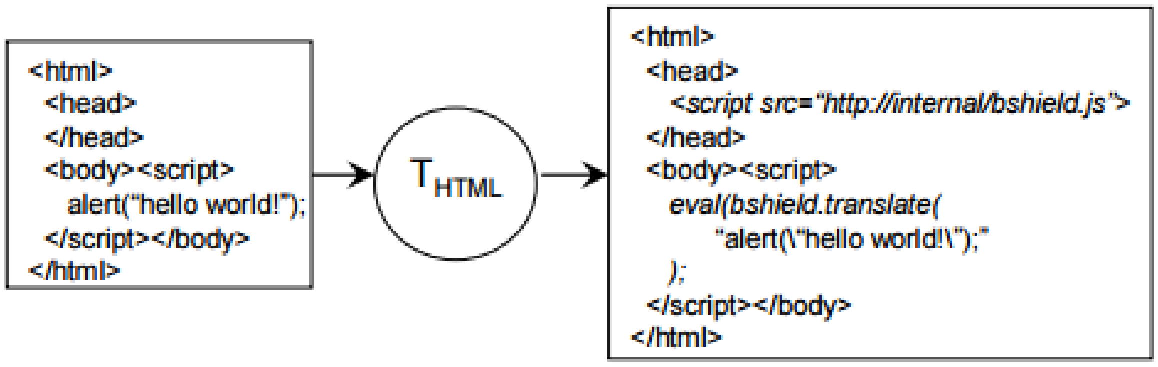


Figure 3.4: BrowserShield injects its own API to the HTML+JS webpage received and replaces vulnerable code with safe alternatives using calls to the BrowserShield API according to the supplied policies (from [2]).

In the first translation T_{HTML} tokenizes the HTML, modifying the page according to the policies enforced and wraps the script elements (see Figure 3.5). In the second translation T_{script} is applied while the page is rendering, parses and rewrites Javascript to access the HTML through a Interposition Layer that acts as a middleware, mediating all access between javascript and the DOM tree (see Figure 3.6). It recursively calls T_{HTML} to any dynamically generated HTML and T_{script} to any dynamically generated javascript code.



```
<html>
  <head>
  </head>
  <body><script>
    alert("hello world!");
  </script></body>
</html>
```

```
<html>
  <head>
    <script src="http://internal/bshield.js">
  </head>
  <body><script>
    eval(bshield.translate(
          "alert(\"hello world!\");"
    );
  </script></body>
</html>
```

Figure 3.5: Example of the translation T_{HTML} being used. `bshield.js` is injected on the DOM and the `alert()` call is replaced with safe `bshield.js` equivalent call (from [2]).

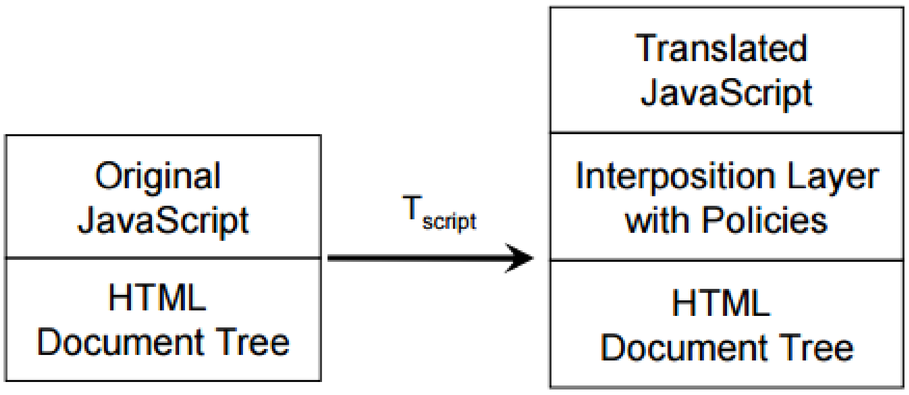


Figure 3.6: Example of the translation T_{Script} being used (from [2]).

BrowserShield adheres to four principles for protecting systems:

- *Complete interposition*: All of the javascript code's access to the HTML is mediated through the deployed middleware.
- *Tamper-proof*: Web pages must not be able to modify or tamper with the Browser-Shield framework in unintended ways.
- *Transparency*: Web page's behavior should not change due to the BrowserShield's framework action, apart from a slight increase on resource usage. The exception being policy enforcement, as if a known vulnerability is found on the Web page and its behavior is changed due to its detection.

- *Flexible policies*: One of BrowserShield's goals is to offer a flexible system that allows its deployment on a number of cases. To facilitate that, there is a clear separation between mechanisms and policies.

3.3 Drive-by download attacks' prevention

3.3.1 CUJO

CUJO (Classification of Unknown Javascript Code) is a system for detection and prevention of drive-by downloads. It is presented as a Web proxy that analyses incoming Javascript code and asserts if the code is malicious; if it is, then CUJO can block the access by the client software to the alleged dangerous Web page [18]. CUJO employs both static and dynamic analysis. Static analysis obeys to the basic principles of a compiler and the original code is decomposed into tokens and analysed. Dynamic analysis is done using a modified version of `ADSandbox` and executed using the `SPIDERMONKEY` Javascript interpreter.

3.3.2 Emulation-Based Mitigation Technique

Another possible solution would be applying emulation techniques similar to those employed to detected shellcode in network streams [19]. Many drive-by download attacks use Javascript to load shellcode into the memory [20], thus preventing this kind of attack would require examining the data retrieved from a Web site and loaded into memory; if shellcode is detected, then there is a good chance that a drive-by download attack is in course.

3.4 Overall problems

The solutions presented that aim to provide a safe environment to run untrusted Javascript are effective at isolating untrusted Javascript code from trusted code, but they cannot prevent drive-by downloads or CSRF attacks by themselves. Any attack done via, unknowingly, accessing a malicious link cannot be prevented by just using the presented solutions. A vulnerability-driven approach would require developing a blacklist of untrusted links or domains, which is impractical and near impossible to be effective, since all it would be needed to go around this blockage was the deployment of a new, unlisted malicious Web site.

The solutions to prevent drive-by download attacks are not 100% effective and cannot ensure complete elimination of this threat [18].

The deployment of an intentionally harmful P2PApp is not the only concern when idealizing our architecture for service sharing. A well-intentioned developer could, unknowingly (i.e., by negligence), leave room on their application for a malicious user to exploit a vulnerability and perform, as an example, a XSS attack to other user in direct contact using the same P2PApp.

Chapter 4

Architecture Description

In this chapter we describe the architecture of the proposed solution, as well as its context and the used language.

4.1 Personal Services

The main scope of this dissertation is the definition and development of secure interfaces for Person-to-Person services. A Person-to-Person service enables the direct communication between two users, which are on different home networks, using a Web application that is running on a browser and does not require centralized control.

The development and deployment of the network backbone, as well as the interface with the supporting network and all of the backend, that is not directly involved with the Web interface, is out of the scope of this dissertation. Despite this, three sample Web applications using the proposed architecture were developed, as a proof of concept, to demonstrate how the final product might behave and look like.

The services can be developed by a third-party, not related to any of the users, and shared within this network. Sharing and using services not developed by known entities is a core concept of this architecture. Since the origin and developer of these services can be completely anonymous, this means that they are intrinsically untrusted and potentially insecure.

Following the analysis of the existing solutions described in the previous chapter, a new solution for safe Web applications should enforce tight control on the Internet connections. In other words, arbitrary links must not work, since these are the main attack vector used on attacks such as drive-by downloads. Although browsers offer a sandboxed environment where Javascript runs, users, as discussed on Chapter 1, often use outdated browsers that might contain known vulnerabilities that, when explored, may allow accesses outside of the browser's sandbox. Because of this, the browser's sandbox cannot be trusted.

4.2 Proposed solution

Since Web applications development became so dependent on the inclusion of Javascript in order to give a fresh look and feel and a high level of interaction, it is notoriously difficult to remove this element without the resulting Web applications becoming outdated and without interactivity. As can be observed in Figure 4.1, to mitigate this effect, instead of

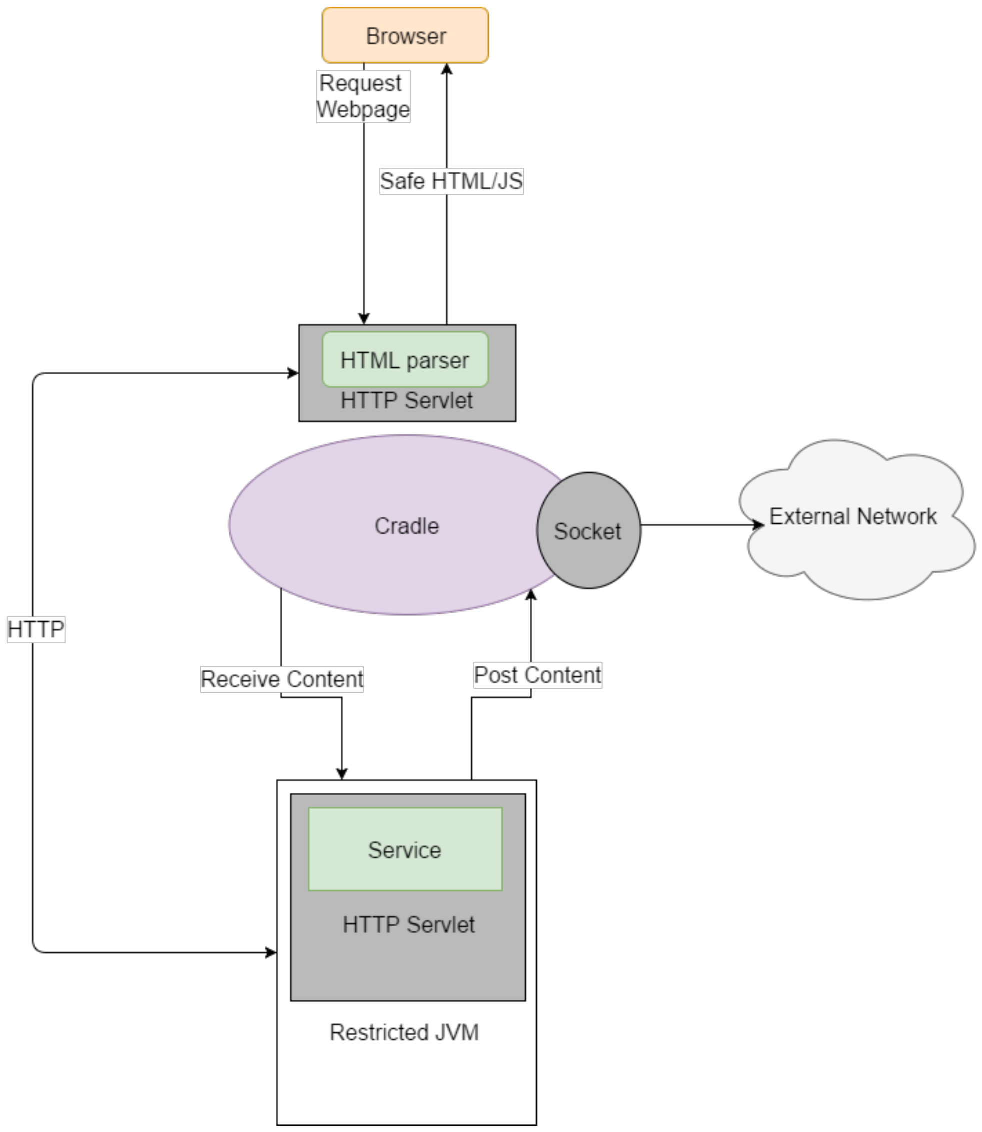


Figure 4.1: Proposed architecture and its components

outright removing all Javascript, the adopted solution uses controlled Javascript. P2PApp developers, instead of using their own arbitrary Javascript code or libraries, write a label or instruction that corresponds to an API call to our own authorized Javascript library. When a Web page is requested by the client, this request is forwarded to a parser that removes these labels and inserts the corresponding API calls. This mechanism works similarly to BrowserShield, described in the previous chapter.

This way, the service developer never directly writes any Javascript, but the resulting applications still use it to provide interactivity and client-side logic.This solution follows the principle of privilege separation. The architecture is broken down in multiple components, each component has its own function, set of privileges and responsibilities, as illustrated in Figure 4.1:

- Parser: The only component directly exposed to the user, responsible for receiving a Web page from the P2PApp, parsing it, replacing the labels with secure Javascript code and sending it to the user. Further details about this component will be given on Section 4.2.1
- Service: The service contains the P2PApp itself, it generates the HTML pages that contain the labels. The Service is technologically agnostic, the demos presented here were written using Java Server Pages, however, any other equivalent technology is valid. Further details on about this component will be given on Section 4.2.2
- Cradle: The only component in direct contact with external networks. All communication to the outside world goes through this component. Further details on about this component will be given on Section 4.2.3

The components on this architecture can be deployed in different machines or can be collocated at will. Mobile users only need to have access to the Parser to access the P2PApps, therefore only this service needs to be accessible to its end user.

4.2.1 Parser

The parser is the element that is in direct contact with the user. Any request to the service goes through it. This parser acts as a proxy Web server to the service backend, it reads HTML with tags and replaces the tags with calls to the internal API written in Javascript (see Figure 4.2). This way, the user's browser never contacts directly with the HTML provided by the P2PApp. Any arbitrary Javascript code is removed before

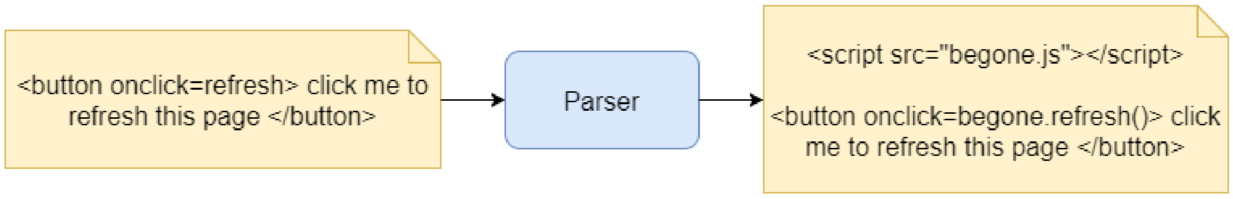

<button onclick=refresh> click me to refresh this page </button>
Parser
<script src="begone.js"></script>
<button onclick=begone.refresh()> click me to refresh this page </button>

Figure 4.2: Simple use case for the proposed Parser: "`onclick`" event's keyword "`refresh`" is replaced with an API call that performs a page refresh. API calls with parameters are also supported.

sending the sanitized P2PApp to the client, as well as any inclusion of external Javascript files (.js) and function calls on event triggers. Only tags that correspond to internal API calls are allowed. Parameters can be supplied to these tags, but since these parameters are considered untrusted, they are properly parsed and escaped removing, all possible attempts to inject code.

This parser is deployed on a Web server. To use it, a client sends an HTTP GET request to the endpoint `parser/access/port/url` where URL is the relative path to the page or file the client wishes to get parsed. The parser then downloads the HTML or file that this URL returns, parses it, and sends the resulting Web page to the client. For instance, if a service is deployed on the URL `http://localhost:8080/service`, the appropriate call to the parser in order to clean the HTML describing the interface of this service would be `parser/access/8080/service`.

While parsing an HTML file, if a `href` attribute is found, the link contained on this attribute will be replaced with a call to the parser using the `/access` endpoint. This way, any CSS file included in the service has to be parsed as well. The resulting webpage that the parser delivers to the user will be always a page devoid of any external links, arbitrary Javascript code or non-authorized libraries.

An alternative solution would be to include the parser within the Service. This would have a performance advantage since, with the adopted solution, two HTTP requests are done for each request on the client, which adds to overhead. However, making the parser an internal element of the Service would compromise on the ability to have the Service a technologically agnostic piece on the architecture.

The parser is also responsible for the injection of the trusted .js files, this includes our API and respective auxiliary files.

4.2.2 Service

This element contains the P2PApp itself. The scope of this dissertation does not set how this service is written, however, for a proof of concept the purposed demos are written using Java Server Pages (JSP) and deployed using Apache Tomcat.

JSP allows the generation of dynamic HTML, similar to PHP or ASP, on the server side. The services run locally on `localhost` using a port defined by the P2PApp programmer and require enough storage to deal with multimedia content. The service is the element that is developed independently and the main origin of security concerns, since all its code is provided by a potentially unknown entity and, therefore, is inherently untrusted.

When a new P2PApp is received from the external network this is the element that is swapped on the architecture.

One important idea when deploying the P2PApp has to be present: the code being run has to be sandboxed. Using the Java programming language, this sandbox is easily achievable by tweaking the Java Virtual Machine with security in mind.

4.2.3 Cradle

This is the only element in the whole system in direct contact with the Internet, outside of the home network. Since this application receives and stores content coming from and going to the external network, there has to be enough available persistent storage for this content. This element of the architecture acts as a software router that receives content from the network and routes content generated from a local P2PApp to a target user running a copy of this same service.

All contacts from the local P2PApp and the remaining components of this architecture with the Internet and any external network is mediated through this component.

This component has to be able to provide an API that allows searching avaliable P2PApps, sharing them and downloading them.

4.3 Initial features available

To develop the API used on the P2PApps it was required to find out what functions were mandatory to be present on that API. A minimal API has to be feature rich enough to provide a dynamic environment for modern Web application development, however, with the least amount of necessary features, code and third party libraries, in order to minimize the risk of security hazards. Bellow, we list the main operations identified, included on the API, that allow the implementation of basic P2PApps. These features are not static, they are merely the absolute minimum required to implement very straight-forward, simple, web applications. Writing the following labels on a HTML page within the service will inform the parser to swap these same labels with "real" javascript code calling functions on the begone.js API:

- refresh: Performs a simple page refresh;
- redirect(URL): Redirects to page on a URL relative to the current page. If, for instance, the current page is on `http://localhost:8080`, calling `redirect(?id=2)` will redirect the browser to `http://localhost:8080?id=2`.
- uploadPhoto(): Uploads a photo from the file system and makes it available to everyone using this P2PApp.
- periodicCall(url, time, function, function arguments): Every given time interval in seconds performs a HTTP GET request to a supplied relative URL. Upon success on this HTTP GET request, its result is passed as an argument to the callback `function` argument. Optionally, if the callback function requires it, extra arguments can be supplied, separated by a comma, which will be passed down as arguments on the callback function;
- updateContent(msg, id): Updates a HTML element with a certain id, appending text contained in the `msg` argument;

- triggerOnKey(key, function, functionarguments): When the supplied `key` is pressed, the function given on the `function` argument is called with the arguments given on the `functionarguments` parameter.
- recordAndSend(link) : This function toggles the recording mode and sends it to the location provided on `link`. When associated with a button or key press, pressing this button or key press starts sound recording using the installed microphone, pressing it again stops the recording and sends the resulting WAV blob to the location on `link`.
- recordAndDownload(): This function has a similar behavior to the previous function, however, instead of uploading the resulting WAV blob, the function downloads it as a WAV file.
- setStorage(key, value): saves the data on `value` on the key in the `key` argument on the browser's `localstorage`.
- getStorage(key): returns the data corresponding to the key on the `key` argument on `localstorage`.

Chapter 5

Implementation

In this chapter we describe the way the architecture is used to implement the demos that illustrate the usage of the architecture to develop P2PApps. We also describe the proposed internal API, its methods and the dashboard, the frontend component to the parser.

5.1 Begone Javascript!

Begone.js is the Javascript file that contains the API calls for all Javascript code that the services use. This file is injected in the Web page when it is being parsed.

The methods contained in this file were conceived with reutilization in mind. The code in this API has to be flexible enough to allow the development of a vast array of different interactive Web applications.

In addition to begone.js, three auxiliary javascript files are included to provide multimedia recording and encoding support, JQuery and Bootstrap are also included. JQuery provides our applications with the ability to perform AJAX requests, it also simplifies our API when it comes to selecting HTML elements. Bootstrap is a widely used framework for front-end development that eases the consistency between the desktop and mobile interfaces. It also provides a modern look and feel to our applications. A P2PApp developer will never use these libraries directly, they will only call internal API methods that use these libraries mitigating eventual nefarious uses that these libraries may provide. One apparent issue with these frameworks is the possibility of these not being secure due to their nature of third party libraries. The complete elimination of this distrust would require the development of a spin-off library written after auditing and removing potential threats. Alternatively, these third-party libraries could be downloaded using their official repositories or Content Distribution Networks. On this last case, there would be a certain degree of trust on code not directly controlled by us, what could constitute a security hazard.

All of these files are injected in the Web page at the parser when a request is made. This way, only these libraries are allowed when developing a service.

5.2 Parser

The parser was implemented as a Java application running on a Web server. Upon receiving a request for a page on a P2PApp from a user's browser it proxies the request through itself to the Service component, where the P2PApp is running, also on a Web server. As stated on the previous chapter, the endpoint to access the parser contains the information required to establish a connection to the P2PApp, the relative link from localhost where it is hosted and its respective port. Using different ports to different P2PApps allows, easily, to deploy multiple P2PApps on the same home network using a single Parser and Cradle.

P2PApp responses to HTTP requests are processed by the parser with `JSoup` to remove potentially harmful code. All of the HTML is processed as a `Document` object, a data type provided by JSoup, which allows us to treat an HTML page as Java Object. This `Document` object contains `Element` objects which will correspond to HTML tags, from these `Elements` we can extract, modify or remove attributes related to the HTML tag these elements correspond to.

The parser will crawl this Document object containing all the Elements forming the HTML page looking for `<script>` tags, which are removed, `href` and `src` attributes, which are processed to just include references inside the `localhost` scope. Also, the parser has access to a list (on a text file) containing all the available event handlers on HTML5, the Parser looks for these event handlers on the Element objects, the attributes on these event handlers must correspond to a tag that can be swapped for a `begone.js` call. To perform this substitution, a dictionary contains a correspondence between tag and the respective API call. If the attribute on an event handler is not recognized as a tag on the `begone.js` API it is deemed untrusted and is removed.

The Parser will recursively parse the arguments on a tag, if that tag has arguments, looking for another possible swap for another internal API call, since it is possible to have nested `begone.js` calls.

If a `<BegoneScript>` tag is found, all of the text between the opening and closure of that tag is swapped to internal `begone.js` calls, if the text inside cannot be recognized as an authorized tag, it is removed. The `<BegoneScript>` has the exact same end as the `<script>` tag in regular HTML, however,only internal API calls are allowed within.

After the document being parsed, cleaned of dangerous `<script>` tags, swapped tags with internal API calls and the arguments of these internal API calls also parsed, the authorized Javascript libraries are injected on the HTML page.

Finally, the resulting page is forwarded to the user's browser, now devoid of dangerous arbitrary javascript.

5.3 Dashboard

As a front-end to the whole architecture, a Web-based dashboard was implemented (see Figure 5.1 for a general view). The main features on this dashboard include launching services, providing a graphical interface to the Parser, registering new users, pairing them and keeping track of their status (if they are online or offline). For demonstration purposes, all the applications developed as use cases of this architecture work in pairs, only one-to-one interactions are allowed. The dashboard is, in practice, part of the parser, and contains methods to interact with it. Since the dashboard is the front-end to the parser, it is the sole element in direct contact with the users.

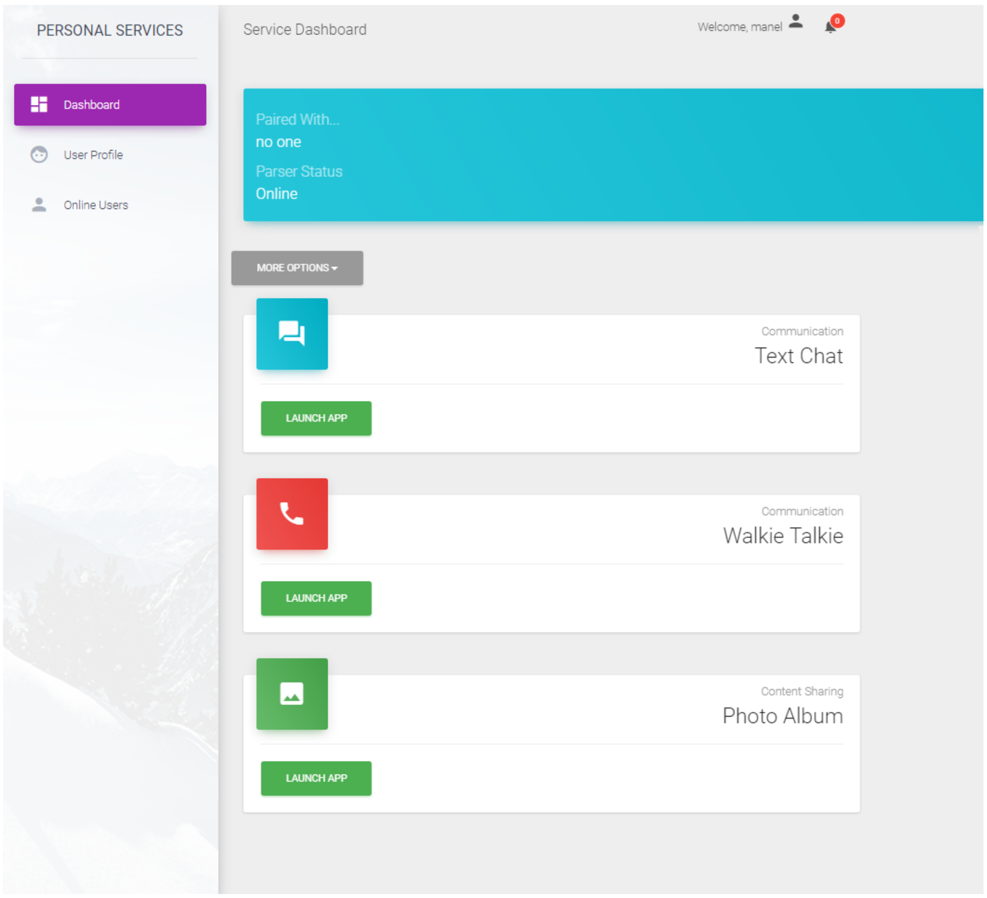


Figure 5.1: Dashboard Overview

When opening the dashboard for the first time a user name must be registered, this registration stores the chosen user name on `local storage` under the key ”username”. This user name registration is also done on the cradle, where not only the user name is stored, but also a timestamp with the time of the registration. The storage is done on a HashMap that is then serialized and written on a file using `MapDB`. This timestamp is updated when the user interacts with the system such as clicking on a button. A Java program running on the same machine as the cradle periodically crawls this HashMap where user names and timestamps are stored looking for a user name which timestamp is older than a certain threshold. If, for instance, this threshold is set to 30 minutes, any user name with a timestamp older than 30 minutes is deleted from the HashMap. The tracking of active users is performed this way.

When a user wants to get paired with another registered user they types the user name corresponding to the person he wants to get paired with. The person that was chosen is notified of the pairing request, unless they was the one that started the pairing process. When both users request a pairing to each other, the pairing is confirmed and registered on the cradle and on `local storage` using the key "pair". After the registration of the pair on `local storage`, the pair is made available within the service and it is possible to use the API provided.

5.3.1 Using the Dashboard

The dashboard loads the latest user registration if possible. The parser status is visible allowing the user to check which of the services are available.

Selecting the "More Options" (see Figure 5.2) menu brings up the form that allows to register a user. If a new registration is done as a different user name, the current identity is swapped to the new name provided in the input box.

The "More options" menu is also used to pair with other users (Figure 5.3).

If a user requests to be paired with another user that did not reciprocated the request, at least at that time, the receiver of the request is notified of that request (see Figures 5.4 and 5.5).

These notifications not only show up momentarily in the interface, they are also stored on a menu, much like on a modern social media Web site (see Figure 5.6).

If a pairing request is reciprocated by the target user, both are notified of their pairing, and the dashboard is updated, showing the new pair's user names (see Figure 5.7 and 5.8).

To check who is available to be paired up, we can navigate to the "Online Users" tab, seen on the side menu (see Figure 5.9).

5.3.2 Mobile user terminals

The dashboard is mobile-ready, adapting to mobile devices. The side-bar is by default hidden; to show it, a user must press the menu icon, this way the interface isn't cluttered with the menu, making for a better experience on small screens (see Figures 5.10).

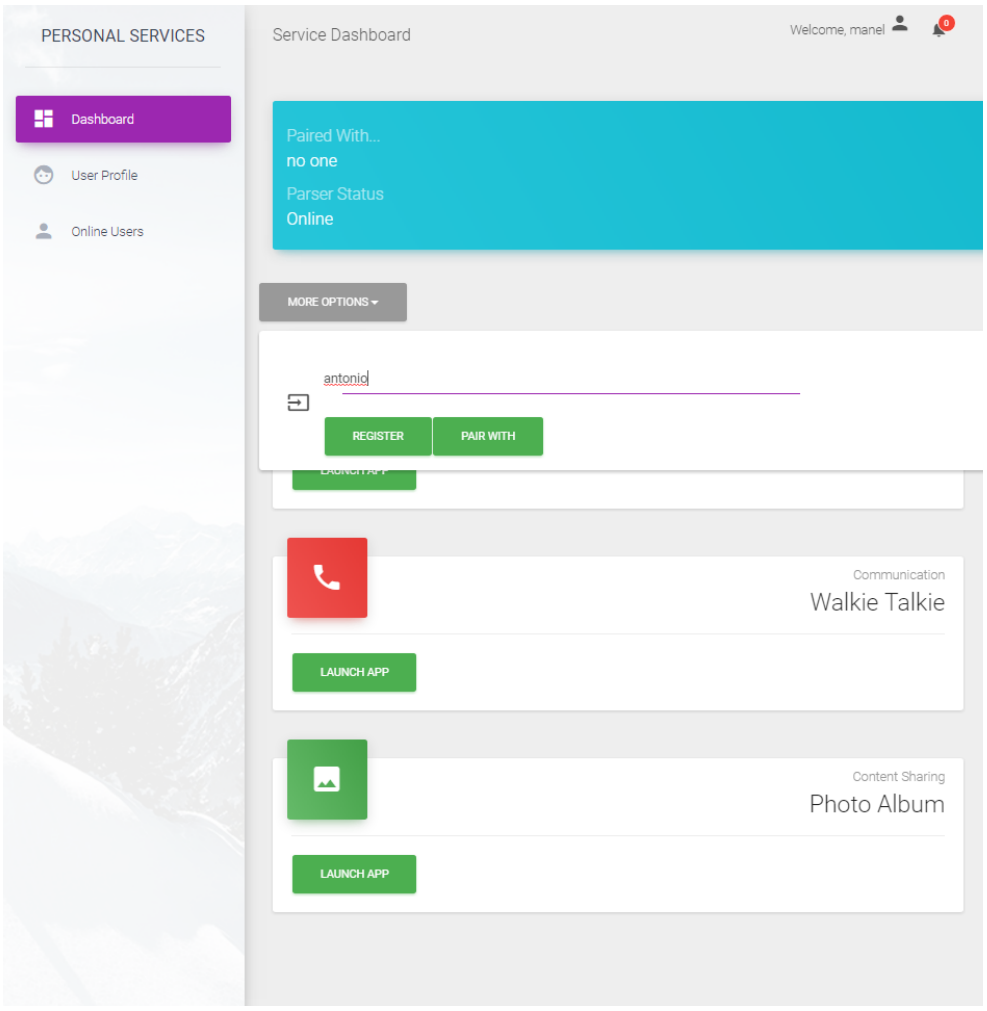


Figure 5.2: Inputting a new user name "antonio" for registration.

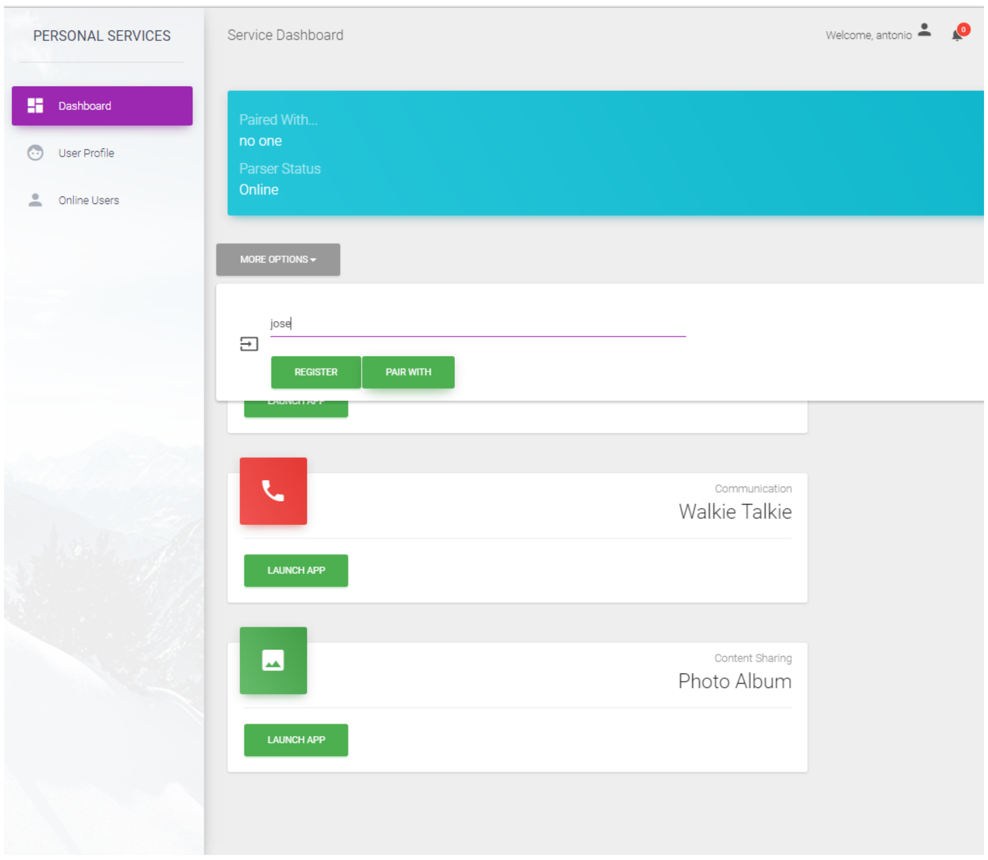
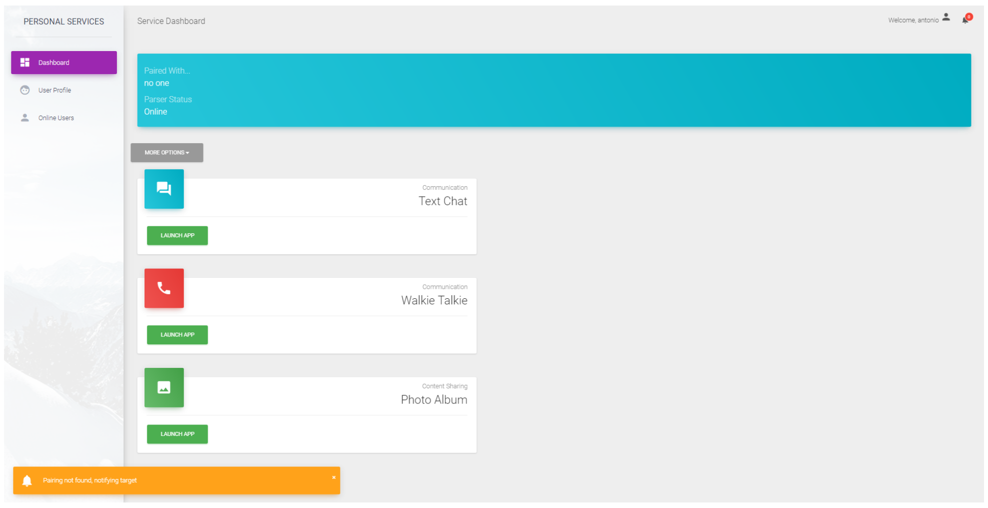


Figure 5.3: Inputting "jose" as user name to pair up with this user.



Pairing not found, notifying target

Figure 5.4: Since "jose" has not requested to pair up with current user "antonio" a notification is sent.

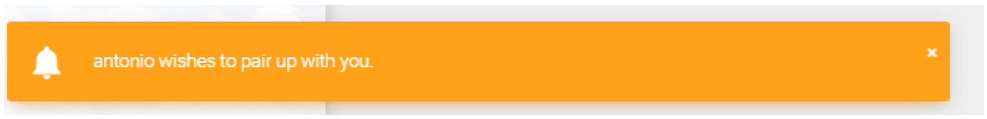
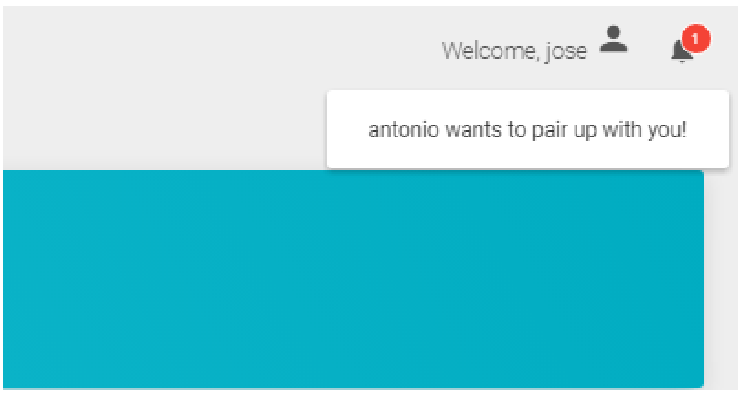
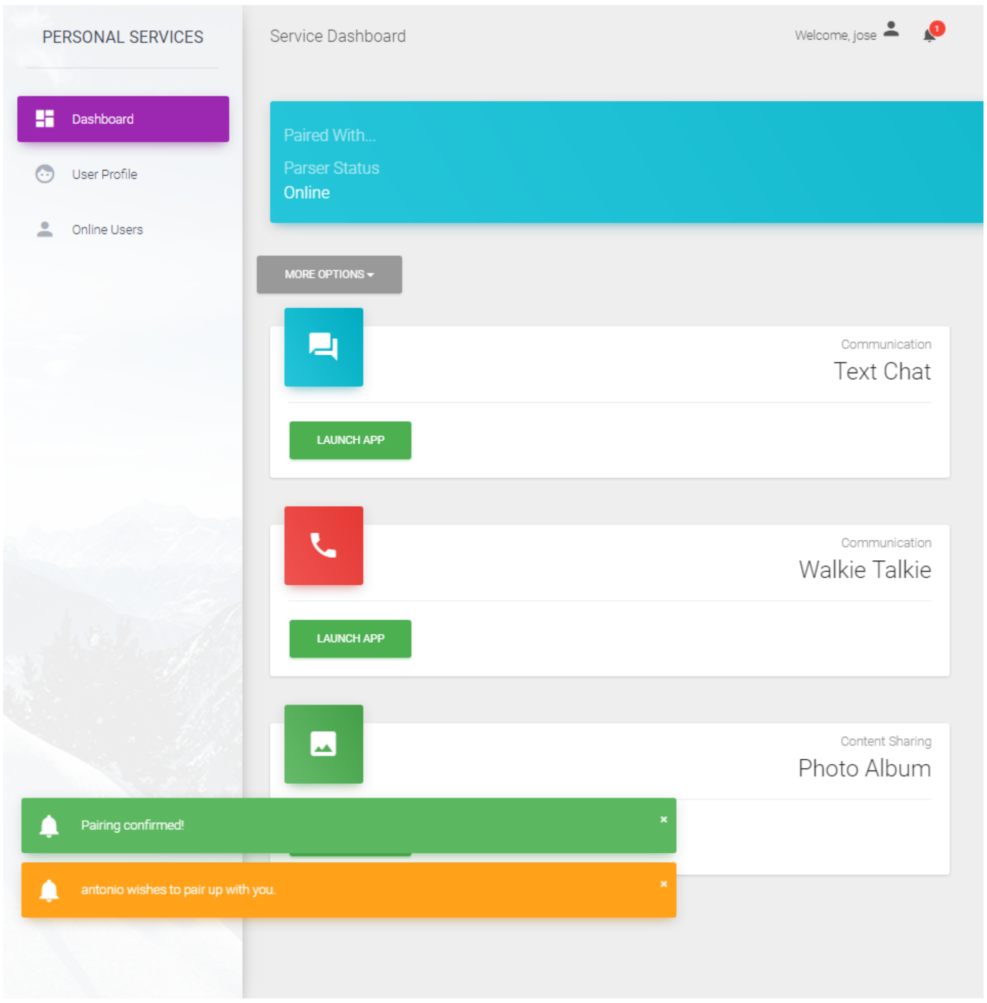

antonio wishes to pair up with you.

Figure 5.5: From the point of view of ”jose”, he has received a notification informing that ”antonio” wants to pair up with him.


Welcome, jose

antonio wants to pair up with you!

Figure 5.6: These notifications are also stored on a persistent menu.



Pairing confirmed!

antonio wishes to pair up with you.

Figure 5.7: Since "jose" is interested in pairing up with "antonio", he pairs up with him back, the green colored notification confirms the establishment of the pair.

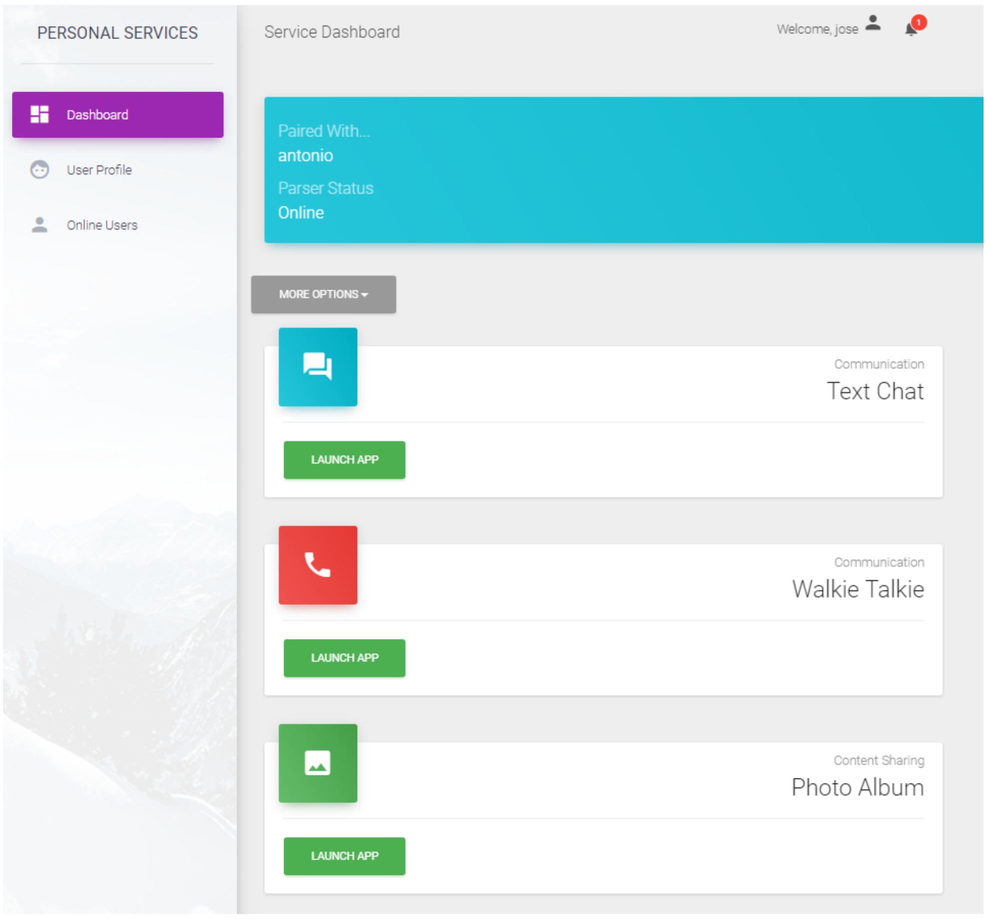
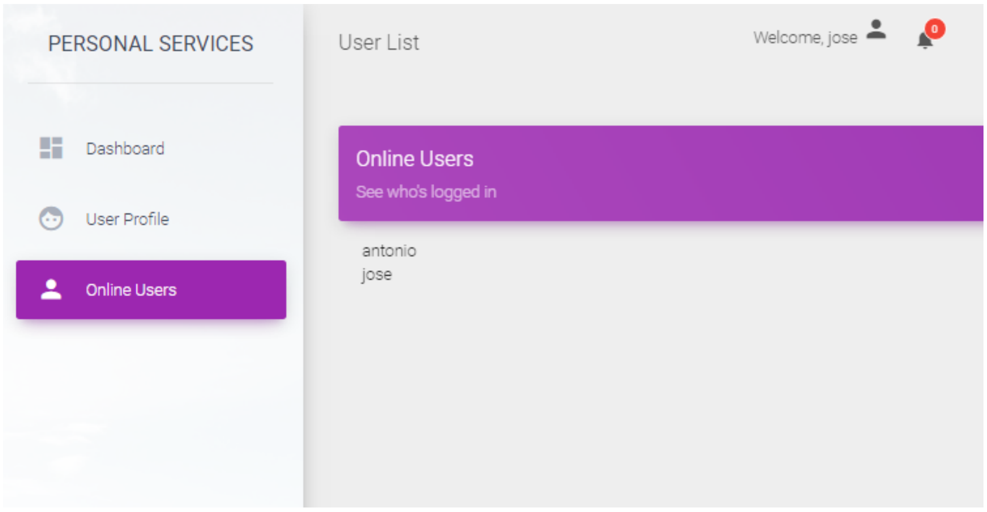


Figure 5.8: The Dashboard interface is automatically updated showing the current pair.



antonio
jose

Figure 5.9: Choosing the "Online Users" tab on the side menu brings up a table with the users currently active.

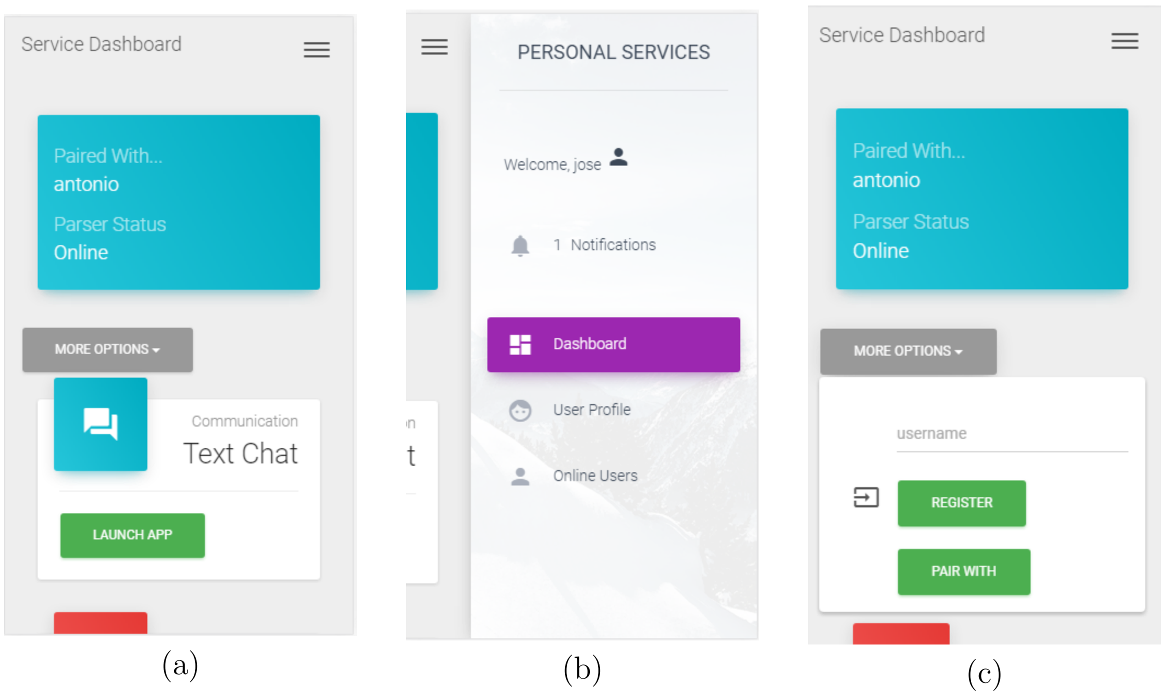


Figure 5.10: (a) Dashboard Overview on mobile; (b) Pressing the menu icon brings up the side menu; (c) All elements, including dropdown menus, are adapted to mobile.

5.4 Demos

In this subsection we will detail our demos, namely their architecture, their GUI, the API methods used and their general flow of use.

5.4.1 Photo Gallery

This first demo shows how the architecture can be used to develop a simple interactive photo sharing service where users can upload their own photos as well as see photos uploaded by other users. Current Web sites like `flickr`, `imgur` or `devianart` offer a Web application capable of receiving uploaded photos from other users and show our own. Figure 5.11 lays out the diagram of a possible architecture for a service of this kind.

Using a browser, a user can upload photos from his own machine using the provided interface. The Parser will receive these photos and redirect them to the P2PApp itself. The P2PApp has its own storage, where the image files are kept. The final destination of the uploaded images is the Cradle, from there, they are forwarded to the external network, ultimately, to another Cradle who will download them and, in practice, do the reverse process. The cradle stores images temporarily until the user requests a page refresh, which will, in turn, request new images from the Cradle.

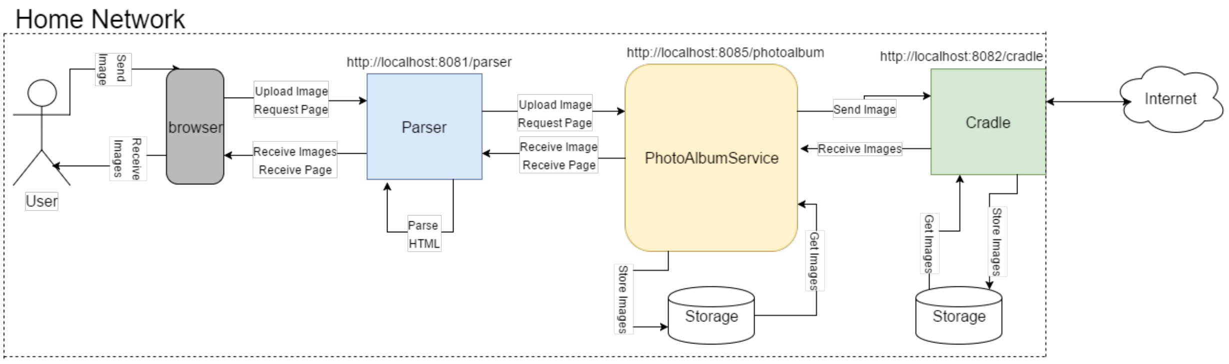

Home Network
User
Send Image
Receive Images
browser
Upload Image Request Page
Receive Images Receive Page
http://localhost:8081/parser
Parser
Parse HTML
Upload Image Request Page
Receive Image Receive Page
http://localhost:8085/photoalbum
PhotoAlbumService
Store Images
Storage
Get Images
Send Image
Receive Images
http://localhost:8082/cradle
Cradle
Get Images
Store Images
Storage
Internet

Figure 5.11: General architecture and use flow of a Photo Gallery P2PApp. All of the interaction with the photo album P2PApp is done through the parser and all of the communication outside of the home network is done through the Cradle application.

The service backend is written using Java Server Pages. Each time the page is reloaded, the service crawls its storage looking for new photos, anytime a new photo is found it is shown on the Web interface as a thumbnail. Clicking on a thumbnail will show a full sized version of the selected photo below.

It is also possible to upload our own photos, when selecting the `browse` button, the file explorer is opened prompting the user to select a photo stored on their own machine, by hitting `upload` the selected photo is sent to the cradle. The cradle then uses its interface with the external network to send it to another user connected to this external network, a copy of the uploaded photo is stored on the persistent storage of the service allowing the viewing of this photo on the gallery. This service is very light on Javascript usage, mainly using it to perform XMLHttpRequests, redirects and forced page refreshes. The logic used to update the Web page's DOM when a new photo is sent to the virtual album is done using JSP. This approach has an immediate shortcoming: every time a new photo is uploaded, for it to be visible on the interface a page refresh is required.

To implement this service, the following API endpoints were used:

- refresh
- redirect
- uploadPhoto

In Figure 5.12 we can observe the proposed interface for a photo gallery P2PApp on a desktop computer and on a mobile device.

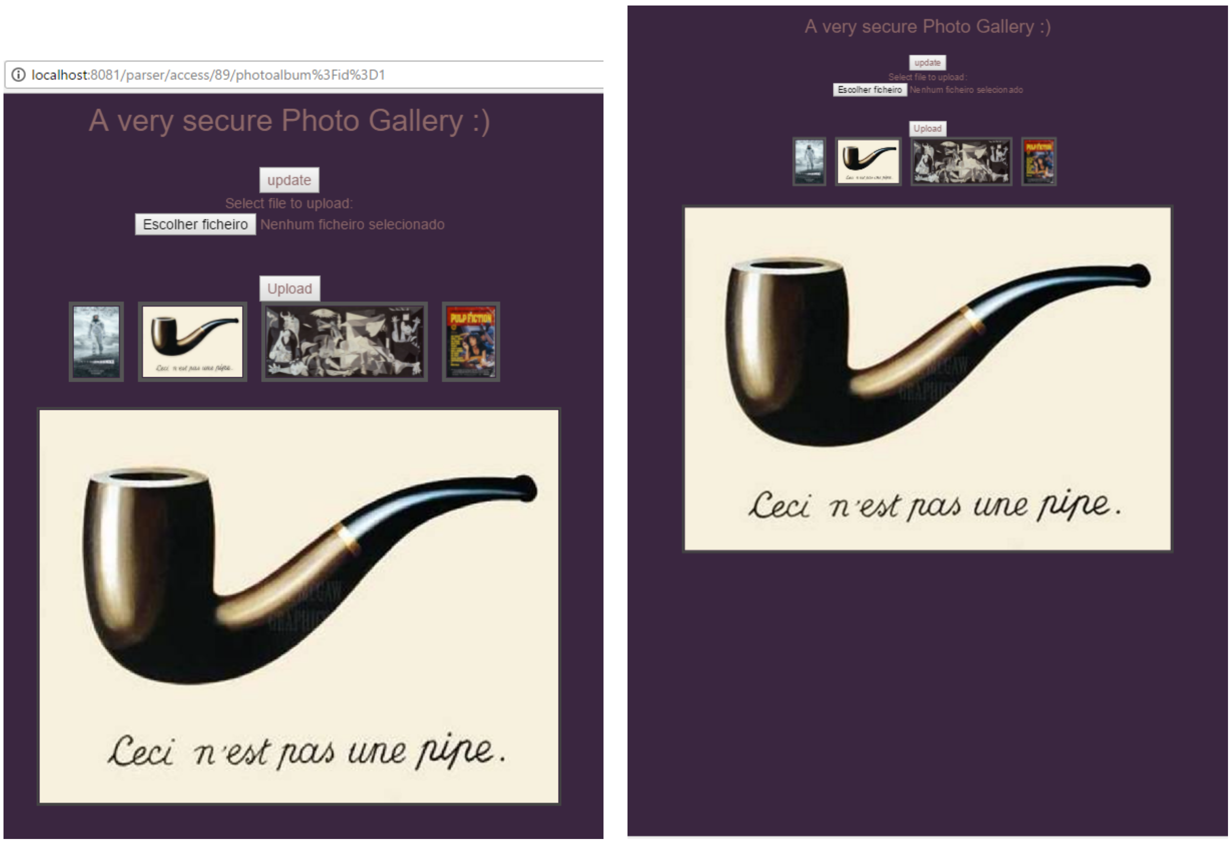

localhost:8081/parser/access/89/photoalbum%3Fid%3D1
A very secure Photo Gallery :)
update
Select file to upload:
Escolher ficheiro
Nenhum ficheiro selecionado
Upload
Ceci n'est pas une pipe.
A very secure Photo Gallery :)
update
Select file to upload:
Escolher ficheiro
Nenhum ficheiro selecionado
Upload
Ceci n'est pas une pipe.

Figure 5.12: Photo Gallery interface on a Chrome browser (left) and on an iphone 6 plus (right). Clicking on a thumbnail will feature the selected photo in a bigger resolution bellow. Uploading a photo from the user's own device is also possible, a new uploaded photo will show up as a new thumbnail.

5.4.2 Text Chat

Going beyond simple Web applications, Javascript is heavily used to perform real-time communication. The ability to text in real time using only a browser, without the need to install a dedicated client application is one of the flagship features of the modern Web. Web applications like `What's App` or `Facebook Messenger` are widely used nowadays, instead of dedicated chat clients such as `MSN Messenger` (now deprecated). Given this, it seemed logical to make the development of a text chat a milestone to prove the feasibility and flexibility of this architecture. In Figure 5.13 we can observe a possible architecture for a P2PApp with this goal.

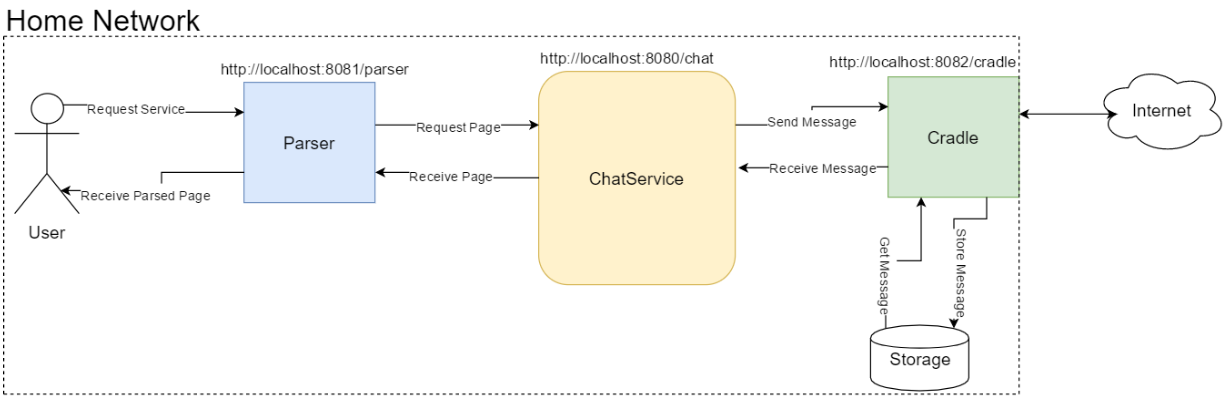


Figure 5.13: General architecture and use flow of a Text Chat P2PApp. All of the interaction with the chat application is done through the parser and all of the communication outside of the home network is done through the Cradle application.

This demo is heavier on the use of Javascript, not relying on page refreshes to update content.

To send a message, a user types it on an input box element in the interface; then, hits the `enter` key to send it. This message is sent to the cradle, that transmits it to the external network. As expected from a typical text messaging Web application, the chat box containing all messages exchanged has to be updated. To do this, one begone.js function allows the controlled addition of new information to an HTML element. This is done when a user is sending a message or receiving new messages.

To receive new messages, a periodic XMLHttpRequest is done, polling the cradle for new messages. If a new message is available, the HTML element containing the chat box with the message history is updated. To achieve a better illusion of real time communication, the polling interval is set to a low value, around one second.

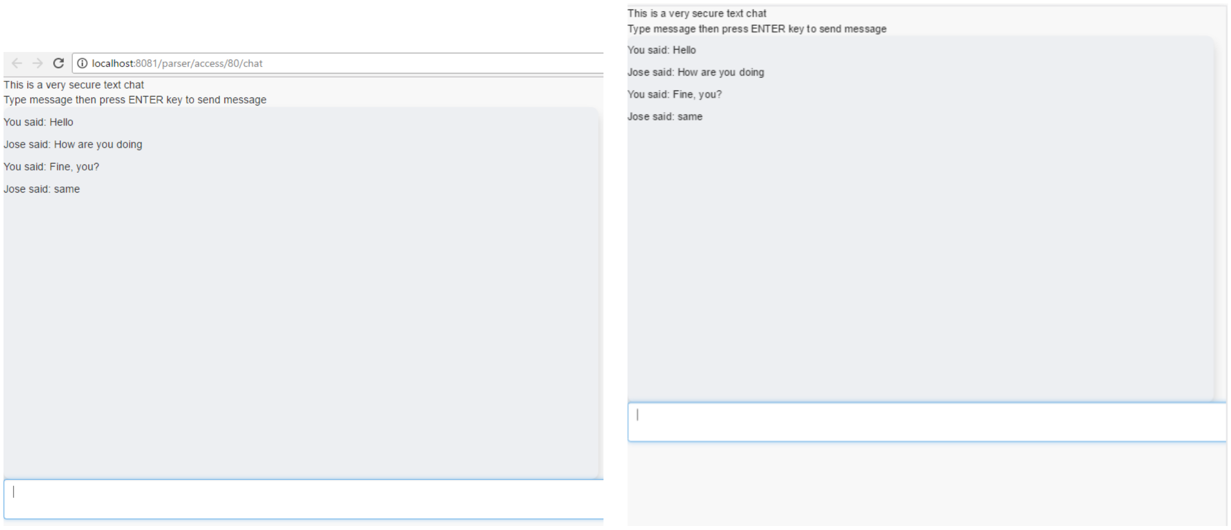
In this demo, Javascript is used to perform the periodic HTTP Requests using jquery's `AJAX`, to trigger events on a key press and to update HTML elements. Since what is displayed on the updated HTML element is user supplied, this API end-point, like all API end-points that deal with user content, sanitize all input preventing XSS attacks.

Like on the previous demo, the parser redirects the inputs of the user to the Cradle, The Cradle stores messages from the Internet and sends them to the interface when the API function `periodicCall` is triggered.

To implement this service, the following API endpoints were used:

- periodicCall
- updateContent
- triggerOnKey

Figure 5.14 shows the proposed interface for this P2PApp on a desktop device and on a mobile device.


localhost:8081/parser/access/80/chat

This is a very secure text chat
Type message then press ENTER key to send message

You said: Hello

Jose said: How are you doing

You said: Fine, you?

Jose said: same

This is a very secure text chat
Type message then press ENTER key to send message

You said: Hello

Jose said: How are you doing

You said: Fine, you?

Jose said: same

Figure 5.14: Text Chat interface on a Chrome browser (left) and on an iphone 6 plus (right). Writing a message on the input box below and pressing the enter key will send it to the paired user. Periodically the application will fetch new messages and display them.

5.4.3 Voice Chat

The capture of audio and video on the Web has been, for years, dependent on browsers' plug-ins or the use of `Flash` or Microsoft's `Silverlight`. Javascript allows the access to peripherals and the capture of multimedia. The increasing popularity of the video and audio chat features on applications such as `Discord` or `Facebook Messenger`, implies that this architecture must also provide room for more advanced and cutting edge Web applications. This demo was developed for this purpose.

This voice chat application does not work like the audio chat on the Web applications mentioned above, where real-time audio communication works a lot like a regular phone with a persistent connection that enables both ends of the connection to always hear one another. To keep the architecture straight-forward and simpler to implement, we adopted a model more akin to how a handheld transceiver, more commonly known as a ”Walkie-Talkie”, works. A button is pressed to start and stop recording, when the recording is stopped the resulting audio is sent to the target user on the other end of the network.

This demo is meant to show how this architecture works when dealing with communication using multimedia, audio in this case, and access to peripherals. Figure 5.15 shows the proposed architecture for this demo.

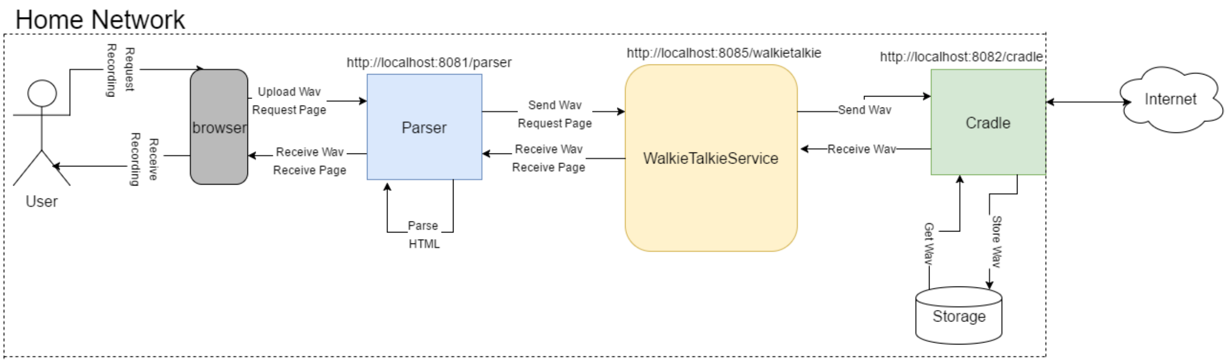


Figure 5.15: General architecture and use flow of a Voice Chat P2PApp. All of the interaction with the application is done through the parser and all of the communication outside of the home network is done through the Cradle application.

Upon finishing a recording, the resulting WAV blob is sent to the Parser and redirected to the Cradle, from there, it is routed to its destination through the Internet. When a WAV blob is received from the Internet, the Cradle stores it and, when requested, it is made avaliable on the interface.

This demo relies heavily on the `getUserMedia` API, which regulates the access of Javascript code to peripherals able to capture multimedia content, such as cameras and microphones.

The access to the microphone and camera is only allowed on HTTPS and localhost connections. Since this service is deployed on the localhost, this is not an issue.

To start a recording, the user presses the button ”record”, a red circle on the button provides visual feedback that tells that is recording, when the user is done he must press again the ”record” button. The resulting recording is encoded as a WAV blob and then sent to the other user we are communicating with using an HTTP POST request to the cradle, which forwards it to the other end of the network.

The `<audio>` HTML element is loaded with the most recent received recording. To update this element with the most up-to-date recording a page refresh is required.

To implement this service, the following API endpoints were used:

- recordAndSend
- recordAndDownload
- refresh

- periodicCall

Figure 5.16 shows the proposed interface on a desktop and mobile device.

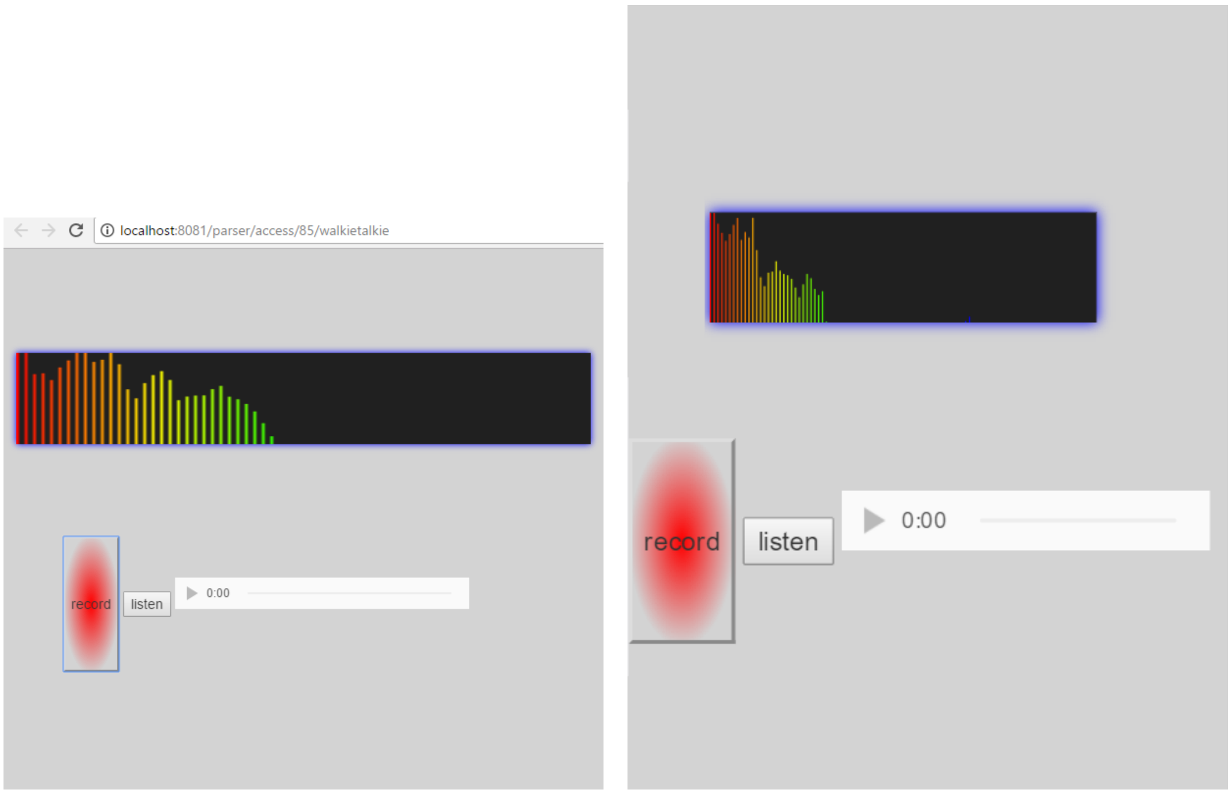


Figure 5.16: Voice Chat interface on a Chrome browser (left) and on an iphone 6 plus (right). The red circle within the "Record" button indicates that recording is taking place, which can be activated by pressing the "Record" button

5.5 Discussion

All of the demos are functional, work and perform as expected. They were deployed on a test implementation of the proposed architecture. The implementation includes a single Cradle that communicates with all peers on the network instead of the real world solution that entails a Cradle for each home network. Only one-to-one communication is allowed, users are paired and communicate with each other. All the elements of the architecture are deployed on a single machine that provides a Web interface to users on the same home network as this machine. Access to the P2PApps can be done through a desktop or a mobile device.

5.5.1 Preventing Common Attacks

Accessing external links from the Internet is not possible through the developed service interface. Any `href` link is only called through the parser. The parser accesses links using a relative path from `localhost`. This way, any link that the parser accesses always begins with `localhost`, and any link included on an HTML file is replaced with a parser call. This makes it impossible to access resources outside of `localhost` scope. Since CSRF attacks mostly rely on the ability of a Web page to access an external link, this type of attack is also prevented.

All of the API functions are written in a way that any content is, by default, untrusted, regardless of the user. With that in mind, any text coming from a user is escaped and cleaned.

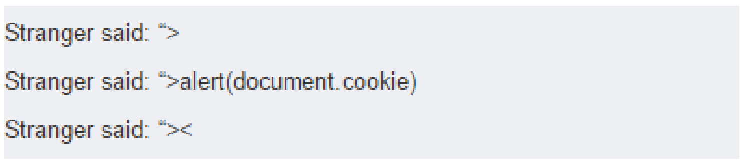
Typical attempts to bypass `<script>` tag filtering do not work such as:

```
"><s"%2b"cript>alert(document.cookie)</script>
```

```
"><ScRiPt>alert(document.cookie)</script>
```

```
"><<script>alert(document.cookie);//<</script>
```

Using `JSoup` on the parser to escape malicious text on the HTML pages, as well as filtering `<script>` tabs, it is possible to deny any attempt to send carefully crafted text containing malicious code, such as the attack seen on Figure 5.17. Using the parser to clean arbitrary Javascript will avoid situations such as the one shown on Figure 5.18. Despite an alert window, by itself, not being a critical security hazard, the ability to show an undesired alert shows that potentially any Javascript code can be run, including code capable of performing more serious attacks such as the ones described on Chapter 2.



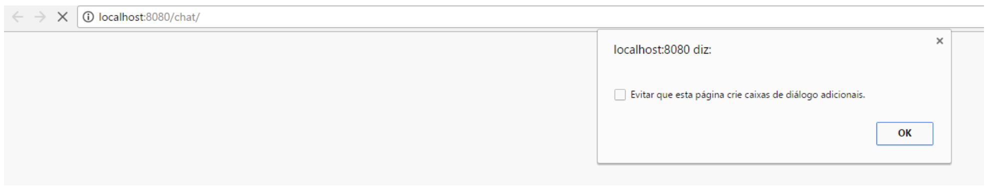
```
Stranger said: ">
Stranger said: ">alert(document.cookie)
Stranger said: "><
```

Figure 5.17: Malicious user "Stranger" on the Chat service attempts to perform an XSS attack using the command `\><<script>alert(document.cookie);//<</script>`. However, the parser is able to detect the attempt and clean it not showing the alert.

On all of the demos developed, attempts to embed and run arbitrary javascript code on the user side are prevented. Deploying a malicious version of these demos with malicious Javascript code on the HTML describing the interface result in no damage on the user side, with the parser successfully filtering `<script>` tags.

The text chat demo features the possibility of receiving and sending user originated text. Since the author of this text is unknown it is inherently insecure and it is treated as such: all text is escaped and cleaned.

Initially we identified two scenarios that could lead to security issues: a malicious user misusing a service or a malicious developer deploying an intentional harmful application. The results obtained imply that these scenarios are correctly prevented. Contents from users or P2PApp developeres is untrusted and never directly presented on the user's browser.



localhost:8080 diz:

Evitar que esta página crie caixas de diálogo adicionais.

Figure 5.18: A malicious version of the Chat service is deployed, if the access is done without the Parser, the code included in `<script>` tags is run.

Since all images `src` attribute entries are cleaned on parsing, the classical approach to inject code using this attribute does not work.

5.5.2 Shortcomings

The development of P2PApps to this architecture is inherently limited. Developeres are not allowed to use any Javascript library they may want to, denying one of the most attractive sides to the development with Javascript: the sheer amount of third party libraries available. Developing P2PApps for this architecture requires a higher degree of effort on the developers' part since they have to make the best out of the available functions and can't use any third-party libraries that may solve certain problems in a more straight-forward way.

Along the development of the demos presented on this dissertation, some difficulties were felt organizing the API with functions that were both useful and highly reusable. There is a clear trade-off between usability, ease of development and security. Offering more features and support to more Javascript powered technologies implies a higher volume of code that need to be micromanaged in order to keep a completely secure, exploit-proof API. Therefore, we predict that the adoption of this solution will produce simpler, less feature-rich applications with very defined purposes.

Drive-by download based attacks are not directly prevented using this architecture. The way these attacks are mitigated is by not allowing the download of arbitrary content (WAV blobs generated using the microphone, for example, is allowed). Effective prevention of this threat, while allowing the development of services that enable the upload of arbitrary content (such as an hypothetical P2PApp that enables version control) would require the analysis of the content being uploaded. In the case of executable files, it would imply the analysis of its behavior or memory inspection. However, the solution here presented can be integrated with a third-party anti-malware solution that would tackle the analysis of binary files.

Although this architecture aims to demand as little trust as possible, such thing is impossible since any developers using this architecture to deploy applications need to trust the provided API and remaining elements on the architecture.

Chapter 6

Conclusions

On this dissertation we designed and implemented a system to develop secure Web interfaces for a person-to-person service infrastructure. These interfaces are written using HTML and Javascript and, as our research revealed, Javascript is a language with a great potential for a variety of Web based attacks that were overcome with our approach.

Following the analysis of the existing solutions we reached the conclusion that none of them, by themselves, at least, can solve the problem of having a completely secure Web interface. Some of those solutions attempt to isolate untrusted Javascript from HTML pages containing sensitive information, such as private information on a social media Web sites. Those solutions, however, do not directly deal with the problem of the untrusted Javascript itself making unsolicited remote requests that may lead to the exploitation of a specific browser security issue and compromising the whole system. Although the same-origin policy should prevent to a great degree remote access to Web sites and services owned by malicious users, this policy is very frequently relaxed to ease the development of Web applications that fetch content from all over the Internet. Furthermore, if a benign service is compromised by an attacker, this can upload malicious payloads containing nefarious code on the (previously) benign service's machine, which would make the uploaded malware to have the same origin has the now compromised service. In this case, the same-origin policy is unable to prevent a user from getting infected by visiting this Web site.

Our solution contains a parser that proxies requests to an untrusted Web application, effectively cleaning arbitrary Javascript code. The usage of Javascript is heavily regulated, using only code contained on a provided internal API. This API is called using special tags that are swapped with real Javascript code containing a call to the correspondent internal API call.

This parser sits on an architecture that has the Web applications (P2PApps) running locally, and all communication with the Internet is highly regulated through a Java program that routes information between home networks. Communication using arbitrary links is not allowed. Since the communication with the Internet is highly regulated, attacks that entail accessing remote links are prevented.

Although not directly prevented, downloading arbitrary executable, binary files from these P2PApps is not allowed thus preventing drive-by download attacks.

The defined internal API proved to be enough to develop simple, straight-forward useful web applications using a combination of simple API calls and the usage of Java Server Pages to generate dynamic HTML.

6.1 Future Work

As already mentioned, the provided API is very limited and to support more complex Web applications it would be required to implement more features. Which means that the base API is always subject to change, addition of new features, removal of obsolete ones and edition of code that can be regarded as dangerous.

The parser itself can be developed to process more complex actions, making this parser more than just swapping keywords with API calls, which, given enough complexity, can become a full-fledged programming language. A possibility is the usage of a tool such `ANTLR` to develop a grammar able to parse commands or the usage of an already existing grammar, such as `JSON`. Any command written could be described as a `JSON` or as a set of nested `JSON`s.

For testing purposes only one-to-one interactions were implemented. As is usual on the Web, most text and multimedia chats allow many-to-many interactions. To do this, it is required the definition of concepts such as rooms.

Since the scope of this dissertation was mostly the development of a secure interface for person-to-person services, the implementation of the suggested architecture, with the exception of the Parser, is not strictly defined. The services can be written and deployed with any technology, as long as they return HTML interfaces. In all of the tests performed, the architecture deployed is not the same as idealized: there is only one Cradle that mediates all of the interaction between all of the registered users and paired users. Ideally, this architecture would allow the interaction of multiple cradles among themselves. To achieve this, it is required to develop an API for external network access.

Bibliography

[1] A. Barth, A. P. Felt, P. Saxena, A. Boodman, N. Baum, E. Kay, C. Jackson, M. Perry, D. Song, and D. Wagner, “Protecting Browsers from Extension Vulnerabilities,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security.* Chicago, Illinois, USA: ACM, 2009, pp. 73–84. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-185.html

[2] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, “BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML,” in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, Washington, 2006, pp. 61–74.

[3] “The 2016 Duo Trusted Access Report - The Current State of Device Security,” 2016.

[4] W. Paul, N. Ben, C. Kavitha, W. Scott, and H. Kevin, “Internet Security Threat Report 2016,” vol. 21, 2016.

[5] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, “Caja Safe active content in sanitized JavaScript,” Tech. Rep., 2007.

[6] “Google Code Archive - Long-term storage for Google Code Project Hosting.” accessed: 2017-03-30. [Online]. Available: https://code.google.com/archive/p/browsersec/wikis/Part2.wiki#Same-origin_policy

[7] J. Ruderman, “Same-origin policy - Web security — MDN,” accessed: 2017-03-30. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[8] S. Nurja, M. Hymavathi, and S. A. Haq, “Evaluation of Web Security Mechanisms using Vulnerability &amp; Attack Injection,” *International Journal of Scientific Engineering and Technology Research*, vol. 5, no. 34, pp. 7035–7038, 2016. [Online]. Available: http://ijsetr.com/uploads/143625IJSETR11835-1232.pdf

[9] N. Bielova, “Survey on JavaScript Security Policies and their Enforcement Mechanisms in a Web Browser,” *The Journal of Logic and Algebraic Programming*, vol. 82, no. 8, pp. 243–262, 2013. [Online]. Available: https://pdfs.semanticscholar.org/700f/f67023a87d4a7ee0363559780209544cd448.pdf

[10] A. Klein, "Cross Site Scripting Explained," 2002. [Online]. Available: www.SanctumInc.com

[11] V. L. Le, I. Welch, X. Gao, and P. Komisarczuk, "Anatomy of drive-by download attack," in *Proceedings of the Eleventh Australasian Information Security Conference - Volume 138*. Adelaide, Australia: Australian Computer Society, Inc., 2013, pp. 49–58. [Online]. Available: http://dl.acm.org/citation.cfm?id=2525483.2525489

[12] D. Yu, A. Chander, N. Islam, and I. Serikov, "JavaScript Instrumentation for Browser Security," *SIGPLAN Not.*, vol. 42, no. January 2007, pp. 237–249, 2007.

[13] M. Ali, R. Shea, J. Nelson, and M. J. Freedman, "Blockstack: A New Decentralized Internet," 2017. [Online]. Available: http://blockstack.org

[14] L. Liu and G. Yan, "Chrome Extensions: Threat Analysis and Countermeasures," in *In Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[15] N. Carlini, A. P. Felt, and D. Wagner, "An Evaluation of the Google Chrome Extension Security Architecture," in *Proceedings of the 21st USENIX Conference on Security Symposium*. Bellevue, WA: USENIX Association, 2012, pp. 7–7.

[16] A. Felt and D. Evans, "Privacy Protection for Social Networking APIs," *2008 Web 2.0 Security and Privacy (W2SP'08)*, 2008. [Online]. Available: http://www.cs.virginia.edu/felt/privacybyproxy.pdf

[17] S. Maffeis and A. Taly, "Language-Based Isolation of Untrusted JavaScript," in *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE, 7 2009, pp. 77–91. [Online]. Available: http://ieeexplore.ieee.org/document/5230484/

[18] K. Rieck, T. Krueger, and A. Dewald, "Cujo," in *Proceedings of the 26th Annual Computer Security Applications Conference on - ACSAC '10*. New York, New York, USA: ACM Press, 2010, p. 31. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1920261.1920267

[19] M. Egele, E. Kirda, and C. Kruegel, "Mitigating Drive-By Download Attacks: Challenges and Open Problems." Springer, Berlin, Heidelberg, 2009, pp. 52–62. [Online]. Available: http://link.springer.com/10.1007/978-3-642-05437-2_5

[20] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending Browsers Against Drive-by Downloads: Mitigating Heap-Spraying Code Injection Attacks," in *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 88–106. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02918-9_6