HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

Ville Rantala

Trust Origin and Establishment with JavaScript Applications

Master's Thesis
Espoo, September 11, 2009

Supervisor: Sasu Tarkoma Professor, Helsinki University of Technology
Instructor: Janne Jalkanen Lic.Sc. (Tech.), Nokia Oyj

HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences
Degree Programme of Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author: Ville Rantala
Title of thesis: Trust Origin and Establishment with JavaScript Applications

Date:	September 11, 2009	**Pages:** x + 71
Professorship:	Data Communications Software	**Code:** T-110
Supervisor:	Sasu Tarkoma Professor	
Instructor:	Janne Jalkanen Lic.Sc. (Tech.)	

Applications written with Web technologies are a growing trend. Web technologies include the JavaScript programming language which has become popular due to its support in modern Web browsers. Today JavaScript is also used to implement installable stand-alone applications in addition to Ajax-style programming. An example of such stand-alone applications are widgets that conform to the W3C Widgets 1.0 specification.

Security is a key concern with these kind of applications because they often have an access to sensitive and valuable information through Web or platform interfaces. One of the main challenges is to determine how to establish trust towards an application. Applications can be benevolent or malicious, but the difference is hard to tell by an end-user. Digital signatures and certificates have been used to help end-users in making a trust decision and to delegate trustworthiness evaluation to trusted parties. These mechanisms have drawbacks that make application development, distribution and adoption more difficult.

In this thesis a new trust establishment mechanism is proposed that helps to deal with the drawbacks. It is based on the Domain Name System and utilizes the originating domain of applications. An implementation of the proposed mechanism is provided on top of the W3C Widgets 1.0 specification and the implementation is evaluated against design requirements. The new mechanism is recognized to bring many benefits to the different parties of the widget ecosystem.

Keywords: JavaScript, Web, Security, Usability
Language: English

TEKNILLINEN KORKEAKOULU
Informaatio- ja luonnontieteiden tiedekunta
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Ville Rantala
Työn nimi: Trust Origin and Establishment with JavaScript Applications

Päiväys: 11. syyskuuta 2009 **Sivumäärä:** x + 71
Professuuri: Tietoliikenneohjelmistot **Koodi:** T-110
Työn valvoja: Professori Sasu Tarkoma
Työn ohjaaja: Tekniikan lisensiaatti Janne Jalkanen

Web-teknologioiden avulla toteutetut sovellukset ovat kasvava trendi. Yksi merkittävä teknologia on JavaScript-ohjelmointikieli, jonka suosio on kasvanut Web-selainten myötä. Nykyään Ajax-tyylisen ohjelmoinnin lisäksi JavaScript-kielellä tehdään myös itsenäisiä asennettavia sovelluksia. Yksi esimerkki asennettavista sovelluksista on JavaScript-sovellukset, jotka toteuttavat W3C Widgets 1.0 -spesifikaation.

Tietoturva on tärkeässä osassa mainittujen sovelluksien tulevaisuuden kannalta. Usein sovelluksilla on pääsy arvokkaaseen ja arkaluonteiseen tietoon joko Web- tai ajoalustarajapintojen kautta. On tärkeää pystyä selvittävään, kuinka luottamus sovelluksia kohtaan syntyy ja mihin se voidaan perustaa. Sovellukset voivat olla toteutettuja haitallisiin tarkoituksiin, mutta loppukäyttäjän voi olla vaikea erottaa niitä vaarattomista sovelluksista. Digitaalisia allekirjoituksia ja varmenteita on käytetty todentamaan sovellusten alkuperä ja täten auttamaan käyttäjiä tekemään valintoja tai valtuuttamaan luotettu taho arvioimaan sovellusten luotettavuutta. Niiden käyttäminen tuo haittapuolia, jotka vaikeuttavat sovellusten kehittämistä, jakelua ja käyttöönottoa.

Tässä diplomityössä suunnitellaan, toteutetaan ja arvioidaan vaihtoehtoinen tapa perustaa luottamus. Ehdotettu menetelmä perustuu Internetin Domain Name System -nimipalvelujärjestelmään. Siinä luottamus perustetaan sovelluksen alkuperään verkkotunnuksen perusteella. Ehdotettu menetelmä on toteutettu laajennuksena W3C Widgets 1.0 -spesifikaatioon ja sen todetaan tuovan etuja monen sovellusten ekosysteemiin kuuluvien tahojen kannalta.

Avainsanat: JavaScript, Web, Tietoturva, Käytettävyys
Kieli: Englanti

Acknowledgements

First I would like to thank Markku Ranta for giving me an opportunity to write this thesis during my work at Nokia. A big thank you goes to the supervisor of this thesis Sasu Tarkoma and to my instructor Janne Jalkanen. They gave very valuable feedback and helped me to during the writing process.

In the planning phase I was inspired by a positive feedback from my colleagues Olli Immonen and Pasi Eronen. Besides them I want to thank Anssi Kostiainen, Teemu Harju and Matti Vesterinen for reviewing and commenting my output.

Finally I would like to thank my family for supporting me. Especially my wife Mimmi and my daughter Tuula who stood by me and motivated me to get this work finished.

Espoo, September 11, 2009

Ville Rantala

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Web applications are a growing trend. Data moves from local storages to online services so that it is ubiquitously accessible. Applications that handle the data can access people's and companies' sensitive information. In many cases, the information is so valuable that it is worth stealing or making unavailable. Information unavailability can cause damage to the parties that are dependent on that information. News about Web attacks are not uncommon. Examples include the Gaza conflict cyber war and the Twitter Web service vulnerabilities. Figure 1.1 shows how the amount of reported incidents has developed over the time.

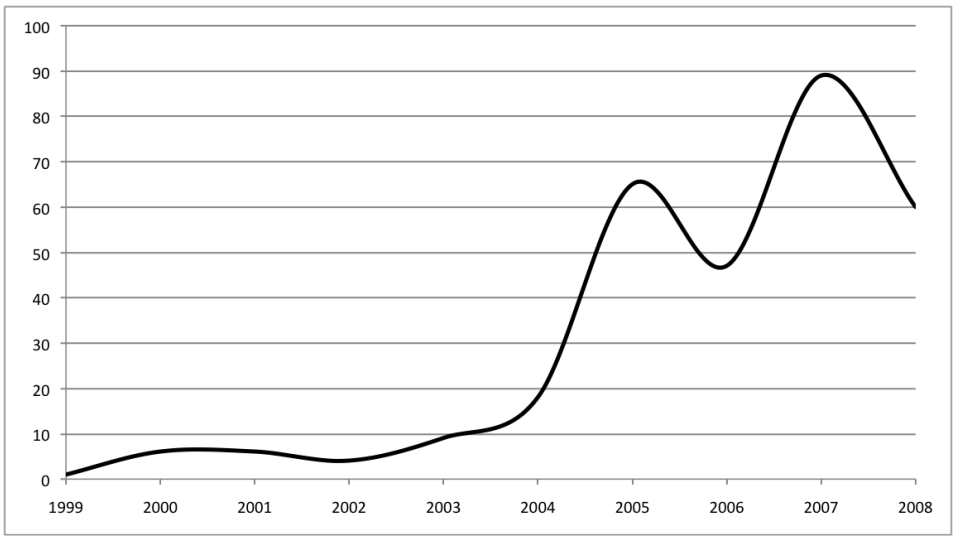

Figure 1.1: The amount of Web application vulnerabilities per year extracted from the web hacking incident database [2]

Incidents can have numerous root causes, but one important aspect is how can Web application users trust the application they use. Trust establishment in Web browsers works implicitly. Users allow JavaScript applications to run on their computer when they browse the Web. Certificates and digital signatures have been used for trust establishment with installable applications. Then the trust decision can be delegated to a trusted party, which has a process in place to verify applications on behalf of the users.

Besides the popularity of applications, the amount of different applications is growing. Application stores like the Apple App Store or Nokia Ovi Store have thousands of applications in their catalogues. Many of them are developed by individual developers rather than software companies. This is a trend that creates different kind of requirements for application development and distribution. Digital signatures have proven to have limitations in a such environment. This is true, especially in the case of JavaScript applications that have a strong Web presence, that use data from multiple sources and that can change their behaviour dynamically. The scope of this thesis is to study trust establishment and to design and implement a better trust establishment mechanism for JavaScript widgets that comply with the W3C Widgets 1.0 specification [5].

1.1 Problem statement

The problem is that applications acquired by users can be malicious. The majority of users are not experienced enough to recognize malicious applications as it would require deep knowledge about the used technology and the ability to examine the application in question. That is why the application trustworthiness analysis must be delegated. Today digital signatures are used to present statements about the trustworthiness. If a trusted party has verified an application, the user can assume it to be benevolent and not to behave maliciously.

Digital signatures have their drawbacks especially when the amount of signed applications is large and when the applications can change dynamically. This is often the case with JavaScript applications. The properties of the JavaScript programming language make applications dynamic. Digital signatures also have a tendency to centralize trust authority to individual companies. Consequently, these companies have lots of control in the content distribution and adoption phases. In this thesis, the trust establishment is studied and a new proposal is made. The proposal tries to help in the problems that exist with the digital signatures. The main research question

is:

How can trust be established to provide usable security?

The word usable in the question covers usability aspects from different angles. The usability can be looked from the end-user, developer or content distributor perspective. The security of a system can be designed robustly, but its usability might then suffer. Design is often a trade off between usability and security [6]. Digital signatures work quite well if only the security aspect is looked at. The usability of them can be considered inadequate for today's needs as discussed in chapter 7.

1.2 Structure of the thesis

The thesis begins with a chapter about JavaScript and Web technologies to describe the foundation for JavaScript applications. After that in chapter 3 the browser security model limitations are discussed and solutions that try to fix those limitations are introduced. Browser security model is studied in detail because the same engine used in browsers is often used as the engine for other JavaScript runtimes. Some of those runtimes are presented in the next chapter along with their way to establish trust. Each runtime has a different mechanism for the establishment and how that is shown to the end-user. In chapter 5, a design for a trust establishment mechanism for a selected JavaScript runtime is described. The selected runtime is a runtime based on the W3C Widgets specification. After the design comes the implementation chapter. The methodology to evaluate the implementation is to compare it against the design principles and other criteria in the chapter. This is done in the evaluation chapter, chapter 7. Finally, a conclusion is presented in the last chapter.

Chapter 2

JavaScript and Web technologies

JavaScript programming language history starts from the Web browser. JavaScript language and other Web technologies are introduced in this chapter to form the basis on which other JavaScript runtimes are built. The security model implemented in current Web browsers is described at the end of this chapter.

2.1 JavaScript history

The JavaScript language has become very popular due to its almost universal support in the Web browsers today. JavaScript programming language was originally developed by Brendan Eich while working at Netscape. The browser scripting language was first named Mocha, later LiveScript and finally renamed to JavaScript. JavaScript was announced in a press release in December 1995 by Sun Microsystems and Netscape. Netscape Navigator 2.0 was the first browser which supported JavaScript. It became available in March 1996. In the same year Microsoft released Internet Explorer 3.0 which also featured similar a scripting language called JScript.

Scripting language standardisation began soon after Netscape and Microsoft had released their browsers. ECMA International started to develop a standard based on JavaScript and JScript and other related technologies in November 1996. The first edition of the ECMAScript scripting language standard was adopted by the ECMA General Assembly in June 1997.

According to the first ECMAScript standard, ECMA-262 [7], the language was originally designed to be a client-side Web scripting language. It was to provide a mechanism to enliven Web pages which were mainly static HTML documents at the time. Scripting languages were not the only way to create

dynamic content. HTML 3.0 specification included some elements for styles and dynamic content and different browser vendors had their own HTML extensions. Extensions could be used, for example, to play sounds, loop images and create scrolling texts. Besides HTML, there were also other mechanisms for enlivening Web pages, such as Virtual Reality Modeling Language (VRML) and Java applets. In the early days of JavaScript, Java applets and other technologies got more attention, but today JavaScript has overtaken them in popularity. [8]

Dynamic HTML (DHTML) is the term that is used to describe the technologies that can be used to create dynamic and interactive Web pages. DHTML was supported in Internet Explorer 4.0 and the same kind of methodology is in use today. [9] DHTML includes the usage of HTML, JavaScript, Cascading Style Sheets (CSS) and Document Object Model (DOM). HTML is used to represent the structure of the resources. JavaScript offers ways to implement the control and the logic of the applications. DOM is the interface that is used from the JavaScript code to access HTML document content. CSS is used to create the layout and styles for the application. Even though DOM and CSS are not a part of the scripting language itself, they are important when talking about JavaScript application security. For Web developers they are a seamless part of the whole Web application development.

2.2 Document Object Model

Document Object Model (DOM) is a platform-independent and language-neutral way to represent structured documents as object models. DOM is used to represent HTML and XML documents as a collection object in a tree format. DOM defines an Application Programming Interface (API) for accessing and manipulating the DOM. It is essentially a programming API for documents.

DOM is today specified by the World Wide Web Consortium (W3C). The support was included in some form already in the early DHTML-featured browsers and the DOM Level 1 specification was recommended by W3C in 1998 [10]. Since then W3C has published DOM Level 2 and Level 3 specifications. Specifications include language bindings, from which ECMAScript binding is the most relevant when talking about JavaScript.

Data that the applications handles usually comes either from the resource document through the DOM API or from a server-side API. DOM is the interface to the document information, but it does not define any access

control model. This means that the access control must be defined and forced in the upper layers of the application.

2.3 Cascading Style Sheets

Cascading Style Sheets (CSS) is used for adding styles to documents (i.e., the visual representation). Styles define the layout of documents and they can be used, for example, to apply fonts and colors. CSS is a W3C specification and the level 1 specification has been a recommendation since 1996 [11]. Style sheets consist of selectors and properties. Properties are used to define what kind of styles to apply and selectors to select the elements the styles are applied to. CSS styles can be included in the documents or they can be stored in separate files. Referring to CSS styles is done either using the *style*-tag or the *@import* declaration.

CSS can be used as an attack vector when inserting malicious JavaScript code to the applications. Microsoft and Mozilla have created their own extensions to the CSS specification that allow scripts defined as CSS properties to execute. For that Internet Explorer introduces *expression*[1] and *binding*[2] properties and Firefox *-moz-binding*[3] property.

2.4 Domain Name System

Domain Name System (DNS) is a hierarchical naming mechanism for clients and servers in the Internet. Data transfers over the Internet are routed based on the IP addresses of the communicating parties and DNS is used to map those IP addresses to meaningful names which are called *domain names*.

An important aspect of DNS is its hierarchical nature and the way how authority over names is delegated. There is no single entity that controls how names are assigned, but the authority is divided into subnames of a domain name.

$$www.some.company.com \tag{2.1}$$

A domain name consists of subnames which are divided using periods. For

[1] http://msdn2.microsoft.com/en-us/library/ms537634.aspx
[2] http://msdn.microsoft.com/en-us/library/ms533503.aspx
[3] http://developer.mozilla.org/en/docs/CSS:-moz-binding

example the domain name 2.1 is divided into four subnames that are also called *labels*. The rightmost part of domain names is called the Top-Level Domain (TLD) which in the example is *com*. TLDs are controlled by the Internet Corporation for Assigned Names and Numbers (ICANN), which is responsible for delegating authority of the subdomains to different parties. The next level in the example domain is *company*. The way how subdomains after this level is delegated is not controlled by the TLD authority anymore. The authority of the *company* subdomain may delegate the next level and so on. Compared to the flat naming scheme, the hierarchical scheme scales better when there are lots of addresses and the administration of the mappings become easier. [12]

DNS plays an important role in the security model that is implemented in the modern Web browsers. It works as the basis of trust, as described in section 3.1.

2.5 Modern JavaScript

In the 1990s when the Web gained popularity also the amount of JavaScript-capable browsers increased. The focus of the Internet changed from a read-only document repository to a read-write application platform. This transformation brought new requirements for Web browsers. New requirements include, for example, higher interactivity, more responsive user interfaces and more personal browsing experience. To fulfil these requirements, more features are required, especially from the client-side of the applications. JavaScript as the client-side scripting language has an important role in modern Web applications, also called Rich Internet Applications (RIA).

The movement from the traditional Web browsing towards RIA required the *page paradigm* interaction model to be changed. Page paradigm is a model where user is forced to load an entire Web page after every interaction. User clicks a hyperlink and a new page is fetched by the browser. This kind of interaction model has limitations when it comes to responsiveness and other requirements of RIA. In JavaScript applications, Ajax-style programming is used to break the page paradigm.

The term Ajax was introduced in 2005 by Jesse James Garrett and it comes from Asynchronous JavaScript and XML [13]. The core of Ajax applications is in asynchronous data loading. It means that data can be fetched from the server asynchronously without interrupting the user experience. This interaction pattern enables RIA written with JavaScript and run in a Web

browsers environment.

The technical enabler for asynchronous data loading is the XMLHttpRequest (XHR) object. XHR is an API that provides functionality for transferring data between a client and a server and it is specified by the W3C [14]. XHR is an important object in the context of JavaScript application security because it is one of the APIs that is used to read and write data and to transfer it over the Internet.

2.6 JavaScript language properties

2.6.1 Security related features

Execution context Every executable JavaScript code runs in an execution context. Every execution context has associated with it a scope chain. Code can only access variables listed on its scope chain. In JavaScript, only function declarations produce a new scope. Every execution context also offers the *this* keyword whose value depends on how the execution context is invoked.

Global object Global object is a unique object, which is created before control enters any execution context. In Web browsers the name of the global object is *window*. Built-in objects and additional environment specific objects are defined as properties of the global object. The global object is referable within the whole execution context.

Global variable A global variable is a variable that is visible in every scope. This means that global variables are accessible and mutable from any execution context. Global object is the container for all global variables.

Run-time evaluation In JavaScript, it is possible to evaluate strings as JavaScript code at run-time. To do this, there is the built-in function *eval*. Run-time evaluation can also be performed with the *Function* constructor and with *setTimeout* and *setInterval* functions.

Dynamic typing In JavaScript type checking is performed at run-time as opposed to static typing where checking happens at compile-time. Variables do not need to be introduced before their use and the type of a variable may vary during the execution.

Prototypal inheritance Object-oriented programming languages with classical inheritance model have the concept of classes from where other

classes can be inherited. JavaScript has prototypal inheritance model which means that it does not have the concept of class and objects inherit directly from other objects.

2.6.2 Script invocation

JavaScript code can be referenced and invoked within HTML documents in many ways. The most common way is to use the *script* tag. It can be used to include code inline (embedded scripts) and from an external source (external scripts). Based on Opera's Metadata Analysis and Mining Application (MAMA) research, about 88% of sites using scripts use it inline with the *script* tag and about 63% from an external source [15].

The second way is to define JavaScript code in event handlers that are tied to HTML tags. For example one could define code that is run when a certain element is clicked with a mouse button. Other not that often used mechanisms are hyperlink URLs prefaced by *javascript:* and, as mentioned in section 2.3, using stylesheet expressions. All these need to be taken into consideration when analyzing application behaviour since any of them might contain malicious code.

2.7 JavaScript security model in browsers

The JavaScript security model that is included in all modern Web browsers is based on sandboxing the executed application. A sandbox isolates scripts from other scripts and from the operating system so that applications are not able to access operating system resources, such as, local file system or the networking layer. Every execution sandbox is associated with one global *window* object which corresponds to a single Web browser window. Windows can also be nested. So from a user's perspective, a single Web page can contain many execution sandboxes.

The primary security policy is the Same Origin Policy (SOP). The way scripts are isolated depends on what DNS domain the page, to which the script is embedded to, is loaded from. That is not necessarily the domain that the script itself originates. [16] The context that the script runs in, and that is defined by the originating domain, is called script's security context. Scripts that run in the same security context can by default access each other's global object when they are embedded in the same page, for example, by using *frame*-tags.

Every global *window*-object has also a reference to a *document*-object that is the DOM representation of the window's content. This means that if a script has an access to a global object, it has also access to the content of that window through the DOM interface.

Table 2.1: Comparing domains when forcing the same origin policy

	URL	Match	Explanation
1	http://example.com/∼username/index.html	-	the reference URL
2	http://example.com/∼another/user.html	yes	only path changes
3	http://example.com:8080/index.html	no	port is different
4	https://example.com/index.html	no	protocol is different
5	http://www.example.com/index.html	no	host is different
6	http://208.77.188.166	no	host is different

When forcing SOP, browsers check that the originating domains of the windows match. Domains match when protocol, host and port values of the domain are the same. Table 2.1 contains examples about the comparison. Host names are matched before the translation to an IP address is done. That is why the example number 6 does not comply to the SOP even if *example.com* would translate to 208.77.188.166. There is an exception to the policy that can be used to make scripts at different domains to execute in the same security context. The *domain* property of a *document* object can be changed to a more general form. This means that for example property value *http://www.example.com* could be changed to *http://example.com* to make these two documents reside in the same security context. [17]

SOP isolates scripts in the client runtime, but also affects on what resources the script can access through the XHR API. Scripts can only make HTTP requests to the domain of their security context. This does not however affect what resources can be loaded to the security context with, for example, *script*- and *img*-tags.

The third concept that's access control is controlled by the SOP is cookies. Cookies are textual data that is stored in the client-side by a Web browser. They can be used to store data about the client and to save state information. Cookies are scoped and the path that they are effective in can be customized. By default cookie scope is limited to the origin of a Web page or to a domain of a security context. Scope can be modified to a fully-qualified right-hand segment of the domain at issue up to one level below Top-Level Domain

(TLD). In practise this means that a Web page from *http://www.example.com* can modify the scope of a cookie to *http://*.example.com*, but not to **.com*. Cookie scope determines can a script access the cookie and is the cookie sent along with HTTP requests to a certain domain. Cookies can be protected from script access by using HTTP-only cookies. HTTP-only cookies are sent with HTTP requests, but they are not accessible from scripts. [18]

The following is an example that demonstrates how SOP operates in practise. Let's assume we have a Web page A from *http://domain.com*. That Web page contains an *iframe*-element whose *src*-attribute point to *http://another.com* which is the URL of a Web page B. A and B have their own execution sandboxes and security contexts since their originating domains do not match. Scripts at A cannot access B's DOM object and vice versa. Only A can open connections with the XHR API to *http://domain.com*. B can only open connections to it's origin. When the browser makes an HTTP request to A's domain, only cookies set in A's context are sent along with the request.

As mentioned, one Web page can contain many sandboxed environments and they can access each other only if their originating domains match. However scripts running in different security context can navigate each others. To be able to navigate means for example changing other frame's location to some other URL. The way this navigation policy is implemented differs between browsers, but the current recommendation that many modern browsers implement is to use so called descendant policy [19]. It means that a frame can navigate only its descendant frames.

Because domain names determine how the policy is enforced, we can say that a domain is a security principal. Principals are entities that can be identified and verified. The process of identification and verification is called authentication. Authenticating in this context means that the browser checks what is the domain name of a Web page. Web browsers were designed as a single-principal platform, which means that different principals are not allowed to communicate with each other and are not able to share resources. SOP takes care that the scripts in different security contexts (i.e., different principals) are isolated from each other.

Chapter 3

Web application security solutions

The previous chapter described the security model in current Web browsers. This chapter discusses limitations of that security model and technologies that are developed to extend the security features of browsers.

3.1 JavaScript security model limitations in browsers

Already in the early days of JavaScript the implemented security model was considered inadequate by researchers [20]. Since then, many research projects have had the goal to improve the security of Web applications in browsers [21, 22, 23, 24, 25].

The way Web has evolved brings additional challenges. When the Web was mainly used to get static documents over a network, the domain-based SOP was applicable. Today the Web is used more and more with applications that collect data from multiple locations and mash it together. These kind of applications are called mashups. SOP makes implementing mashups hard, since it is designed to prevent access to multiple domains. Browsers were designed as a single-principal platform, where as mashups would require multi-principal security models [24]. Another change in the nature of the Web is that today it is not just about Web sites. It is also about data APIs. An API is a data source that is accessible with URL. Web applications must conform to SOP when they want to use those APIs.

SOP is designed to rely on the DNS domains of the resources. The way domains are matched against each other does not always correlate to the entity that provides a resource. For example the first two lines in table 2.1 conform to SOP even if these two resources might be controlled by different

entities. The first one is controlled by *username* and the second by *another*. This is an example of a false positive case. Domain matching can also create false negatives. For example, the last two rows in table 2.1 may very well be the same entity even though the other contains *www* prefix. The prefix causes these two to not conform to SOP. DNS names can also be rebinded for malicious purposes and to subvert the SOP [26].

The root cause of a vulnerability can be in the browser security model or in some other factor or in a combination of these. Other factors are, for example, problems in browser implementations, careless authors, careless users or badly designed authentication flows. Here we concentrate on vulnerabilities that are caused by or related to the security model implementation. The following list is gathered based on the research done in this area and on the Open Web Application Security Project (OWASP) Top 10 list from 2007 [27]. The OWASP Top 10 is a list of the most serious Web application vulnerabilities collected by a world-wide security community. The list contains JavaScript security issues and vulnerabilities.

SOP issues The SOP was designed to isolate applications from different domains from each other. It does not, however, have an affect on from where the data can be loaded with, for example, *script*- and *img*-tags. With them it is possible to define any URL in the *src* attribute, and the resource from that URL will be loaded to current security context. Because JavaScript supports run-time evaluation, it is possible to load scripts from external domains that will run in the context they are loaded to. Besides loading data, the *src* attribute of the various elements can be used to transfer data outside a sandbox. Data can be defined as query string parameters and sent along with a *HTTP GET* request. An example of this data transfer scenario is the URL `http://any.domain.com/path?data=somedata`. Content security policies (section 3.6) have been proposed to make these workarounds more controllable by the page authors.

Cross Site Scripting (XSS) XSS flaws occur when an attacker is able to run code in a security context where it is not intended to run. In practise, this happens when an application renders user generated content to a Web page without properly validating and encoding it. XSS is possible because browsers determine the security context based on the origin of the document and not on the origin of the script itself. XSS can be prevented in the applications, but also more general solutions, such as, browser-enforced embedded policy [28], static vulnerability

detection [29] and dynamic Web application analysis [30] have been proposed.

Cross Site Request Forgery (CSRF) CSRF attacks happens when a malicious Web site causes user's Web browser to perform an unwanted HTTP request to some other site. This is possible when the browser's security policy allows Web sites to send requests to any domain. Typically cross site requests can be more harmful if the trusted site uses HTTP cookies to store the user session. Browsers are implemented so that they send cookies associated with a domain along with a request that is made to that domain, regardless of the origin of the request. Based on the browser security model design, there is no way to tell where a request originates. To accomplish this researchers have proposed an *Origin* HTTP header [31, 32]. Currently, the CSRF vulnerability prevention must be implemented in the application layer above the browsers, because it cannot yet be assumed that the proposed additional header is implemented in all browsers. Both client- and server-side solutions for application level CSRF protection have been proposed [33].

Session stealing Stealing of a user session happens when an attacker can access the session identifier that identifies the session between a client and a server. If session identifiers are stored in HTTP cookies, an attacker may be able to read them programmably with JavaScript or by eavesdropping HTTP traffic. Cookies were designed to enable the implementation of state on top of the stateless HTTP protocol. Today, they are used as the primary mean to authenticate a user in Web applications, but as such, their applicability has been questioned and more secure cookie protocols have been proposed [34]. Browser sandbox security model was designed to prevent access to the user file system. This is why cookies are currently practically the only way to store data on the client side. The limitation causes application developers to use cookies for purposes they were not designed to be used for, such as, authentication and authorization.

Absent content security methods When a document is loaded to the client browser, it is fully accessible to every script that is running in the same security context through the DOM API. If the document contains sensitive data, malicious scripts may be able to read it and compromise the confidentiality, integrity or availability of the data. The SOP protects the content, but because it does not have an affect on, for example, *script*-tag source, sensitive data can be transmitted

outside the current security context to any domain by sending it as query string parameters.

Implicit trust in domains When users browse to a Web page they implicitly trust the party that controls that page. The implicit trust allows scripts in that page to run in the user's computer. The party in control can be authenticated using Hypertext Transfer Protocol Secure (HTTPS) with certificates signed by known Certificate Authorities (CA). But, users browse also pages without this protection.

Integrity issues Browser security model does not guarantee Web page integrity, unless the connection between the server and the client is protected with Transport Layer Security (TLS) or some similar mechanism. A user might browse to a Web page from a trusted domain, but the page might be modified by an attacker before it is rendered to the user. These integrity issues affect the informational content of the page and also the JavaScript code that might be embedded to that page. Attacks of this type are called Man-In-The-Middle (MITM) attacks. Even protection on the HTTP layer is not always enough. It can be ineffective itself [35] and it can be only used to guarantee transport layer security. It is not suitable for ensuring application layer integrity (i.e., has the embedded JavaScript code changed after the trusted author wrote it).

3.2 Public Key Infrastructure

Public Key Infrastructure (PKI) is the infrastructure needed for digital signatures that are applied in the Web. Digital signatures together with certificates can be used for authentication and data integrity. With authentication it is possible to confirm an identity you are dealing with. In the Web, it is used, for example, to identify that a user actually deals with the correct party when making money transactions. Data integrity makes sure that the information related to the transactions stays the same end-to-end and that it is not tampered by unknown parties (i.e., protects from a MITM attack).

The implementation of digital signatures is based on public-key cryptography. It is a form of cryptography in which asymmetric key algorithms is used. Asymmetric key algorithm means that the key used to encrypt a message is different than the one used for decryption. The different key types are called a *public key* and a *private key*. Data that is encrypted with a public key can only be decrypted with the corresponding private key. Also the reverse

is true, data encrypted with a private key can be decrypted only with the corresponding public key.

Data integrity is implemented using public and private key pairs and a hashing algorithm. First a hashing algorithm is used to create a message digest from the data. Message digest is also called a one-way hash because the calculation can only happen one-way. It is not possible to calculate the original data from the hash. Other important characteristics of message digests are that they are fixed length and unique for the hashed data. Fixed length makes them applicable for efficient encryption and uniqueness makes sure that whenever the data is changed the message digest changes too. After the message digest is calculated, it is encrypted with the sender's private key. The encrypted value is the digital signature of the message. When the receiver gets the data, the signature can be decrypted with sender's public key to unpack the message digest value. Receiver can verify data integrity by calculating the message digest from the message and comparing it to the unpacked value. By making this check the receiver can also be sure that the sender of the message has the private key corresponding to the public key used for decryption.

Linking a public key to a real-world entity is done with certificates. Digital certificates are used to claim that a public key belongs to a certain entity. The entity can be, for example, a name of a company or a name of a server. A certificate includes information about the entity in question, information about the issuer and most importantly, the digital signature of the certificate for ensuring that some trusted entity is linked to a certificate. The linkage is checked by going up a chain of certificates and making sure that a trusted certificate is found from the chain. Trusted certificates are called *root certificates*. A common application for certificates in the Web is the Transport Layer Security (TLS) protocol [36]. It is used to provide data integrity and server authentication in a client-server communication over the Internet.

A list of trusted CAs can be found from an operating system level or it can be shipped along with an application. For example Web browsers such as the Firefox comes with a list of trusted root certificates that are decided by Mozilla. Mozilla has a public policy for managing the list of certificates [1].

Because real-world trust relationships can change, there is a need for a dynamic certificate verification. For example a list of trusted certificates hard-coded in a mobile device Read Only Memory (ROM) years ago might not reflect the current state. Also a trusted party may become untrusted, for example, if agreements have been violated or if the party is compromised and

[1] http://www.mozilla.org/projects/security/certs/policy/

the private key is leaked. Then anyone that have an access to the private key can pretend to be the trusted party. Certificate Revocation List (CRL) is one of the mechanisms to handle revoked certificates. A user agent using certificates can be configured to check the certificate status online using a defined CRL location before making a trust decision. Another mechanism is the Online Certificate Status Protocol (OCSP) that is designed to be more efficient protocol [37]. Also OCSP Mobile Profile has been specified by Open Mobile Alliance (OMA) to make the protocol more suitable for mobile environments [38].

3.3 Signed scripts

Signed scripts can be used to identify and authenticate the source of the scripts and to guarantee that it hasn't been modified or tampered since it was signed. When the source of the script is known, it is possible to implement more fine-grained security policies. Scripts can for example be granted different privileges and capabilities based on the origin.

If a script is signed by a trusted entity, such as a well known CA or some other Trusted Third Party (TTP), users can assume that the script does not contain any malicious parts. It can be then granted more privileges than to a script from an unknown source. It is possible for example to define that a system access is allowed to all scripts that have been signed by Nokia or an entity verified by VeriSign. VeriSign is one of the publicly recognized CAs. [39]

Signed scripts are supported by Mozilla-based browsers such as Firefox. It supports signing entire Web pages. Signed resources are placed into a Java ARchive (JAR) file along with the associated signature that has been calculated from the resources. The signed page is referenced with a special URI scheme, for example *jar:http://example.com/secure.jar!/page.html*. Extra privileges that can be assigned to signed scripts include access to browser settings, user browsing history and browser internal APIs. [40]

Signing scripts and Web pages addresses many of the limitations presented in section 3.1. It is still possible to perform XSS attacks, but it is impossible to abuse extra privileges of signed scripts. Signed scripts are isolated from other scripts and one cannot call functions defined in the signed part outside. Implicit trust based only on a domain and integrity issues can be handled with signatures. The trust is not anymore fully implicit because signed Web pages are allowed to run only if they are signed by trusted authority. Trusted

authorities can be predefined in browsers or users can have their own trust chains. Integrity of the pages can be guaranteed in more higher level than using transport layer security solutions, such as, SSL or TLS. Users can validate that the integrity has not been compromised and that the application has not been modified between the signing and the execution.

3.4 Security zones

Microsoft Internet Explorer (IE) has a feature called security zones that can be used to configure different levels of security for groups of Web sites. Web site belonging to a group belongs to a security zone. IE includes five predefined zones: Internet, local intranet, trusted sites, restricted sites, and my computer. All zones are by default assigned to a security level. Security levels contains configurable security options, such as, file access, ActiveX control or scripts and level of capabilities given to Java programs. Some security zones are automatically recognized from the originating domain of a page and some must be configured manually by the user or the system administrator. [41, 42]

Like signed scripts, security zones make trusting domains more explicit. They also allow more configurable and controllable policies. In the traditional browser security model scripts have all or nothing type of rights, but with security zone configurations, users can modify the settings in a more detailed manner.

3.5 XML signatures

XML Signature is a W3C recommendation that defines an XML syntax for digital signatures. It can be used to sign any digital content. When external content is signed the signature is called *detached* and when the signed content is inside the XML signature document the signature is called *enveloped*. Because XML signatures can be used to sign any content, it can also be used to sign for example Web pages. [43]

XML signatures are not meant directly for Web browsers in the same way as signed scripts (section 3.3), but they can be used to implement data integrity validation and source authentication in browser-like environments. For example the W3C Widgets specification takes advantage of the XML signatures [5]. Like signed scripts, XML signatures depend on the PKI.

3.6 Content security policies

Content security policies are policies that restrict or allow content access and that operate on the client-side. With these policies, Web page authors can define what kind of content and from what source can be accessed on that page. Web application authors know what kind of content and what scripts are needed for the application to function properly [28]. Because of this, content security policies can be predefined by the authors and enforced in the browsers.

Examples of content policies include content security policy by Brandon Sterne [44] and content restrictions by Gervase Markham [45]. In both, the main issue they tackle is XSS vulnerabilities. Content authors can define from where scripts can be included to the same security context. That prevents malicious scripts from arbitrary domain to be injected. An example use case from the content security policy documentation is described next. An auction site wants to allow images from any domain, plugin content from only a list of trusted media providers and scripts only from its internal server. With content security policy proposal that use case could be defined with the value *allow self; img-src *; object-src media1.com media2.com; script-src userscripts.example.com* in the HTTP header named *X-Content-Security-Policy*.

Same Origin Mutual Approval (SOMA) policy is a proposed policy for the same kind of functionality [46]. It defines a manifest file that can be used to specify what domains can access resources. The idea in all of the proposals is that the content authors can specify from where a resource can be accessed from. It can be said that the access control decision is a mutual agreement between the content author and the content integrator. Adobe Flash has implemented this using the *crossdomain.xml* policy files [47].

Besides Adobe's implementation, the content security policies are not widely implemented in the modern Web browsers. Recently W3C has been specifying this kind of policy. It is called *Cross-Origin Resource Sharing* [48]. It specifies a syntax to define how resources are accessible from different domains. It helps to deal with XSS and also makes it possible to perform client-side cross-origin requests.

3.7 Server-side proxy

Server-side proxies can be used to allow cross-domain communication with JavaScript applications running on Web browsers. The SOP restricts communication to the domain the application originates, but the originating domain can contain a proxy which forwards requests to an arbitrary domain. It is then possible to create mashup applications that use data from different sources. The author of the application can control which domains can be contacted with policies that are applied in the server-side component.

An example application that uses server-side proxy is iGoogle and other OpenSocial containers [49]. iGoogle is a customizable Web page to where users can embed JavaScript gadgets. Gadgets fetch data using the XHR API, so they are limited to the SOP restrictions. Cross-domain communication is made possible by sending requests through the iGoogle server. Proxy server forwards requests and returns them to the caller gadget.

Chapter 4

JavaScript runtimes

Previous chapters of this thesis have concentrated on the Web browser. In this chapter four other JavaScript runtimes are introduced. Specifically, the application installation process is presented. The installation process is important because that is usually the phase where trust is established and trust decisions are made. Run-time behaviour is discussed later on in this thesis, in section 7.5.

4.1 Apple Dashboard

Apple Dashboard is a runtime for Apple Dashboard widgets. A Dashboard widget is a set of source code files and other resources packaged in a folder that has the file extension *.wdgt*. The main metadata file for the widget is called *Info.plist* where the author can define widget properties and needed capabilities. Capabilities can include things like network access, command line access and the usage of Internet and Widget Plug-ins. The widget can have an access to the file system outside its container and it can call native OS X applications through the plug-ins. This means that widgets can have basically the same level of capabilities as the native applications. [50]

Widget packages can be acquired from anywhere. Apple also provides a repository for widgets where users can search for widgets and download them. During the installation process, an installation prompt is shown to the user. An example of such prompt can be seen in figure 4.1. The prompt does not contain any information about the capabilities that the widget gets after it has been installed (i.e., the ones defined in the widget's metadata file). There is also no information about the origin of the widget. Only the widget's name

and icon are shown. The question mark icon in the prompt opens the generic help documentation for installing Dashboard widgets. The trust decision must be made by the user based on the information that has been available before the installation begun. That information can be for example that it was downloaded from the Apple repository or that the package was received from a trusted friend. Dashboard does not support signing of widgets.

Figure 4.1: Apple Dashboard widget installation prompt

4.2 S60 Web Runtime

S60 Web Runtime (WRT) widgets are JavaScript widgets for S60 mobile devices. They are structurally similar than Dashboard widgets 4.1, but before deploying they are packaged as Zip-archive with a file extension *.wgz*. The process of granting capabilities differs from the way it is handled in Dashboard so that capabilities are queried from the user during the run-time and not during installation. Before installation, one prompt is shown to the user. That prompt is the top left prompt in figure 4.2. If, after the widget is installed and instantiated, the widget tries to access an API that needs extra capabilities, additional prompts are shown. Additional prompts are the ones at the right side of the example figure. The bottom left prompt is the additional information dialog that is shown if the user selects *More info* from the security warning prompt. Additional information dialog lists the capabilities that are granted if the user accepts the request. None of the dialogs show information about the origin of the widget. Signatures are not supported. Consequintly no certification information can be available either.

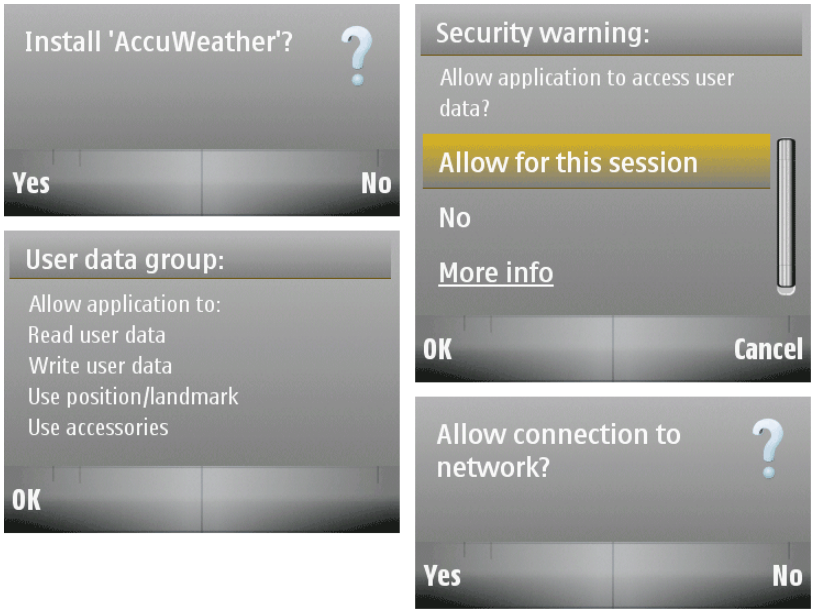

Figure 4.2: S60 Web Runtime widget security prompts

4.3 Adobe Integrated Runtime

Adobe Integrated Runtime (AIR) is a runtime that can be used to run applications that use JavaScript or other Adobe technologies such as Flash. Applications are run on desktop environment on top of the AIR and they can have access to the operating system services through the AIR APIs. An example of such service is file system access. All AIR applications must be digitally signed using either *self-signed* certificate or a certificate issued by a trusted CA [51]. The difference of the used certificate type is shown to the user when the application is installed. In figure 4.3 the dialog at the bottom is shown if a self-signed certificate is used. It says that the publisher (i.e., the source of the application) is unknown and that the application will have unrestricted system access. The top left dialog is an example where the publisher of the application is verified by a trusted CA. The dialog still advices the user to think whether to trust the publisher in question. The top right dialog is an example where an AIR application is acquired from the Adobe AIR Marketplace [1]. It shows the originating domain of the file that will be either installed directly or saved to the computer.

[1] http://www.adobe.com/cfusion/marketplace/index.cfm

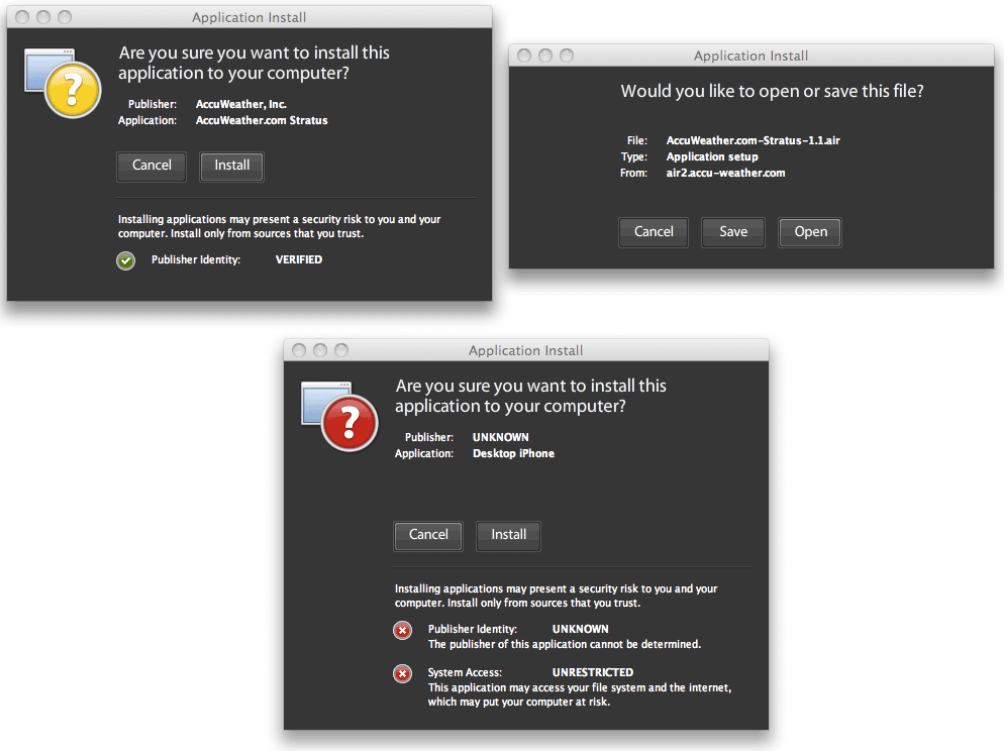

Figure 4.3: Security prompts of a verified and unverified Adobe Air application

4.4 W3C widgets

W3C widgets are JavaScript applications that are packaged according to the W3C Widget Packaging and Configuration specification [5]. The specification is W3C Candidate Recommendation from 23 July 2009. There already exists some widget user agents that support the W3C widget format. An example is an open source Apache Incubator project Wookie [2]. Microsoft has announced that they will support W3C widget in Windows Mobile 6.5 [3] and there exists supporting implementations for Symbian-based mobile devices [4] [5].

The W3C Widgets specification does not specify how security policies are defined or what is the conformation process. However, it does define how widgets can be signed using XML Signatures [52]. It means that imple-

[2]http://incubator.apache.org/projects/wookie.html

[3]http://blogs.msdn.com/windowsmobile/archive/2009/03/18/windows-mobile-6-5-what-s-in-for-developers.aspx

[4]http://www.samsungiq.com/

[5]http://www.betavine.net/

menters of the user agents can include policies that require certifications and digital signature verifications. There is also an Access Request Policy specification that defines how widget network access policy must be defined [53]. Security policies might include things like a widget must be signed using a certificate issued by a trusted CA if it needs access to arbitrary DNS domains. Or that the user is prompted for an access for each domain listed in the widget's metadata.

Chapter 5

Trust establishment design

This chapter first contains general trust and trust model definitions. After that a design of trust establishment for W3C Widgets is presented. The design contains a description about the widget ecosystem and the relevant trust relations. It also contains requirements for a good trust establishment mechanism. These requirements are used to evaluate the implementation which is described in the next chapter, chapter 6. Implementation chapter is followed by the evaluation chapter.

5.1 Trust

Trust is a complicated phenomena that can be defined in many ways and that is studied from many viewpoints such philosophical, sociological, psychological, computer scientific, economic and legal view point. Essentially trust is a directed relationship between a trustor and a trustee which is affected by multiple factors. Trustor is a party that believes that a trustee is trustworthy and therefor trusts the trustee. The factors that influence trust can be classified, as shown in table 5.1. As seen in the table, both subjective and objective properties of the trustor and trustee as well as the contextual properties are influencing trust relationships.

The origin of trust relationships is in real world connections. The connections can be for example social or legal. In computer science these connections must somehow be modelled in a digital way so that they can be processed

Table 5.1: Factors influencing trust [1]

Trustor subjective	Confidence, Belief, Gratification, Disposition
Trustor objective	Goal/Purpose, Regulation, Laws, Standards
Trustee subjective	Benevolence, Motivations, Honesty, Faith
Trustee objective	Security/Safety, Reliability, Availability, Integrity, Dependability, Competence, Utility, Predictability, Maintainability, Reputation
Context	Situation, Risk, Environment

computationally. This process is called trust modelling. Trust establishment in digital world and in the Internet is usually more challenging than in the real life because the communication channel between the different parties is not always secure and the visual trust impression is often missing. An important aspect of digital trust is also the fact that identities might be faked. [1]

Trust evaluation is the technical methodology that is used to determine the trustworthiness of a trustee. In computational trust evaluation different factors of trust are taken in consideration and a trust value is outputted. Trust value can then be used to make trust decisions. Even though trust is a complicated concept, in the digital world the trust decision often ends up to be a binary value that reflects whether the trust is established or not. For example user either allows or does not allow a widget to run.

When a system has to make a trust decision based on a policy or based on a user interaction it has to consider whether some application is trusted by the system. After a positive trust decision the application in question gets capabilities. This does not however guarantee that it is trustworthy. It means that the security of the system relies on the trustworthiness of the application. If it proves to be untrustworthy the security of the system is compromised.

5.1.1 Trust models

Trust model is the method to specify, evaluate and set up trust relationships amongst entities for calculating trust. Trust modeling is the technical approach used to represent trust for the purpose of digital processing. [1] There are many ways to model trust. They can be based for example on the PKI [54], user behaviour [55] or reputation [56].

A common trust model applied in Web browsers and with installable appli-

cations is based on the PKI and trust towards a list of Certificate Authorities (CA). As mentioned in section 3.2, a popular Web browser - Firefox - is distributed with a default list of trusted CAs. The content of the list is decided by Mozilla. When a user downloads and installs the browser, an implicit trust relationship is formed between the user and the CAs that are listed in the default list.

Computational trust modeling can be used to help end-users to make trust decisions. However sometimes the aspects influencing trust are very hard to interpret with the means of computer science. For example, a study [57] showed that people pay more attention to the visual design of a Web site than to more rigorous indicators when they assessed the credibility of the site. The scope of this thesis is the level that can be achieved with computational means.

5.2 Design for W3C Widgets

W3C Widgets specification has been selected as the target for the trust establishment design. The specification is still considerably young and there are not yet many major implementations in a production state. That is why it is possible that there is room for improvements that has not been discovered yet. The specification is getting lots of attention in the industry, which indicates that the content of the specification will make a difference once it gets implemented. W3C Widgets is also a good aggregation of different widget runtime features. It contains properties from many of the popular widget runtimes as described in the widget landscape document [58].

The architecture of the W3C Widgets can be seen in figure 5.1. Parts marked with an asterisk are specified in the W3C Widgets specifications. Other parts are technologies that are used in a traditional Web browser environment.

5.3 Widget ecosystem

The widget ecosystem includes individual people and companies who operate in different roles. Functional roles of the different parties can be categorized as presented in figure 5.2. The source of the figure [4] is about the mobile ecosystem, but the same functional role classification can be used in this context. Starting from the left side of the figure there are content owners. Content owners can be companies or individuals (i.e., user-generated con-

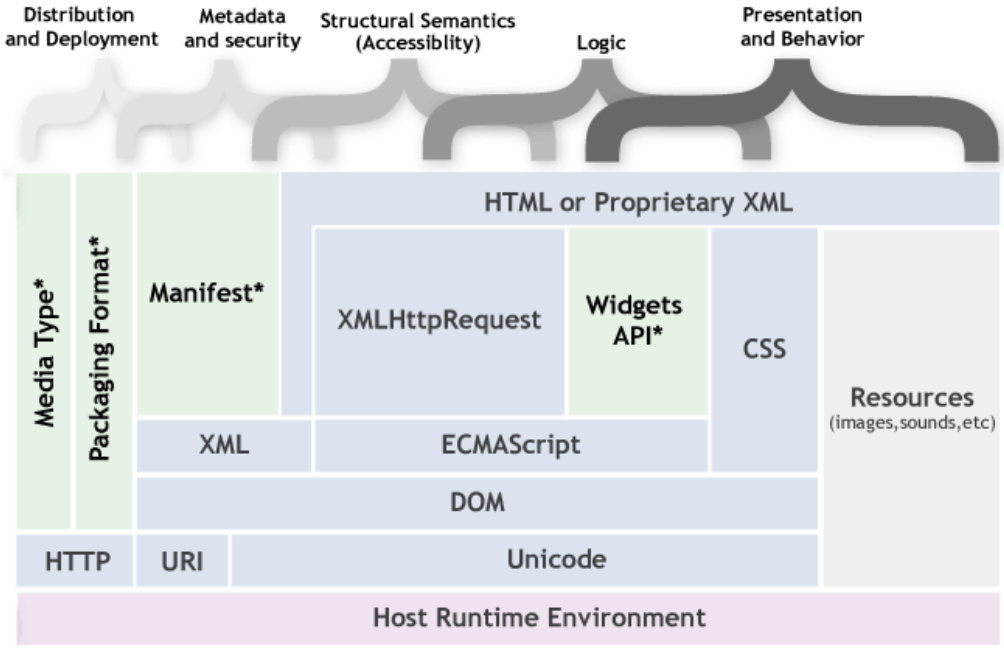

Figure 5.1: The architecture of W3C Widgets [3]

tent). The closer to the right side the role is the closer it is to the end-user.

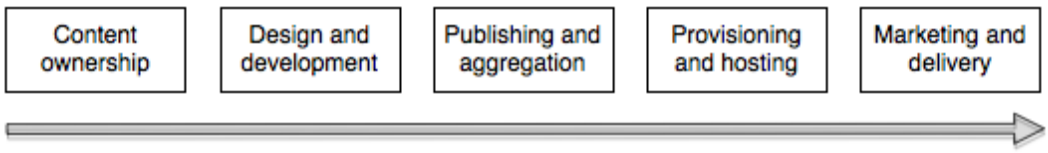

Figure 5.2: Functional roles in the widget ecosystem [4]

In a W3C workshop on security for device access, the use cases for security policies were reported [59]. Those use cases relate to the widget ecosystem, since from there it can be extracted what kind of motivations different parties have. Trust establishment method is closely related to the security policy conformance, so they must be implemented so that the described use cases can be implemented.

Enterprise An enterprise that offers devices to its employees must be able to control the policies.

Delegation Security policies and trust decision must be delegatable. For

example an end-user might want to delegate them to a trusted third party.

Differentiation A party must be able to define security policies themselves in order to differentiate. They might, for example, want to give applications or services a privileged access to certain capabilities.

Portability Used mechanisms should allow portability, so that whenever the used device is changed the policies can be transferred along.

Duplication A user should have the opportunity to have the same applications with the same security configurations on multiple devices.

In this thesis the parties of the widget ecosystem are categorized in the following way. The categorization does not follow closely on the business environment, but rather concentrates how the trust relations between the parties can be described and defined on.

User is the end-user of an application.

Application / Author is the application itself or the author of an application that has the capability to modify it and distribute it to a user through a Distributor. This role is separated to two parts because this role can be trusted based on the application itself (Application) or on the origin of it (Author).

Distributor distributes applications to users, but only controls the distribution channel and not the applications themselves.

Runtime in which the applications run and that is utilized by users. It can be either the runtime manufacturer or some party controlling the runtime policies such as a telecom operator.

Data provider provides data that applications consume to be beneficial to Users, for example, in the case of mashups.

5.4 Trust relationships

In this thesis there are several relevant trust relationships between different parties that belong to the widget ecosystem. The ecosystem and the parties are described in section 5.3 and the trust relationships between them are

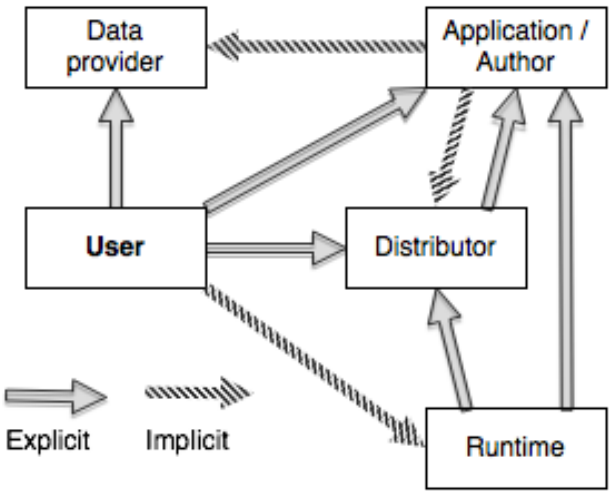

Figure 5.3: Relevant trust relationships between roles in the widget ecosystem

presented in figure 5.3. Solid arrows represents explicit relations that require trust decisions to be made. Dashed arrows represent more implicit relations that are relevant, but are assumed to exist naturally. For example a user is expected to trust the runtime because she/he uses it and has decided to run applications on top of it. Application authors are expected to trust distributor and data providers. Distributor, because that is the only way to get their applications to users and data providers, because authors' application require data from those data sources to be valuable to users.

There are multiple explicit trust relations from the user and the runtime to other parties. This is because they are the end points of the application distribution chain and also the place where trust decisions are concretized. For example the user either installs an application or not, or the runtime either grants capabilities to running JavaScript code or not. Trust decisions can be made by the runtime by counting on the transitivity of trust. In general the transitivity means that if A trusts B and B trusts C then A also trusts C. Conceptually it can be argued that trust cannot be considered transitive [60], but in this context that assumption can be used to ease the user role in making trust decisions and by doing that creating better usability. The delegation of trust is also one of the policy requirements defined in section 5.3.

An example of harnessing trust transitivity is the way Apple distributes applications through the App Store [61]. Users express their trust towards

the runtime by using their Apple devices and towards the distributor by downloading applications from the App Store. App Store takes care of the communication to the application authors. Authors must be a part of a developer program before they can submit applications to the App Store. After submission Apple checks the applications and only accepts the ones that can be downloaded and installed by the users. Consequently, users do not have to make an explicit choice whether to trust an application or its author and they can rely on Apple's judgement and policies.

5.5 Trust establishment requirements

This section lists properties that are required from a good trust establishment implementation. The requirements are gathered based on the information in the previous chapters of this thesis and the policy description use cases described in section 5.3.

Effortless mapping to policies It should be easy to define security policies based on the trust establishment mechanism. Security policies must remain manageable even if they are complex.

User experience Trust establishment should be implemented so that it is easy to represent to the end-user. On the other hand, an end-user must be able to delegate the trust decision to a trusted party, such as, a hardware manufacturer or some other organization.

Developer experience Development and authoring of the applications should be easy. Developers or application authors should not be stressed with additional work or costs. An example of such a cost is the cost of the developer certificates, that are needed to test applications on the runtime they are targeted to run on.

Application updatebility After the application is installed and the trust has been established, it must be possible to update the application without another trust decision making if that is not needed. For example if an application author updates the application logo there is no need to dissolve the established trust.

Flexibility The mechanism should be flexible so that the different parties can act independently and there is no excess processes, which limit or slow down the application adaptation and usage.

Distributor control Application distributors must be able to control the trust establishment. For example, based on the policy description use cases, a distributor (e.g., handset or service provider) must be able to differentiate with trust decision features.

Implementation effort Establishment should be easy to implement and it should be clear enough so that different implementations can interoperate.

Easy to understand Understandability helps with many of the other requirements. Easily understandable trust establishment enhances user experience and helps the other parties dealing with it.

Compatibility with the existing infrastructure The implementation must not create too many new concepts. It should take advantage of the existing Internet infrastructure and enable today's business models and business use cases.

Chapter 6

Implementation of originating domain utilization

This chapter starts with an overview of the proposed trust establishment implementation for W3C Widgets. Before the implementation details, the widget installation process is explained and the proposed modification to the process is described. After that comes the details of two implementation styles. The first one is a complementary implementation to the original widget specification and the second is a proposed alternative that has different properties. The properties of each of the two implementations are compared in the implementation summary section of this chapter. It is followed by user interface examples that show how the proposed implementation could be made visible to the end-user.

6.1 Overview

As described in section 3.1 the current security model in Web browsers is heavily based on the DNS domain or the hostname that a Web page originates from. With installable widgets there might be no connection at all to any domain. The widget is a stand-alone package that might be distributed with a physical media, like USB flash drive, and that does not acquire data outside its package. An example is a JavaScript game. In case of games, there might be no need to connect to any DNS domain during the installation or usage.

The idea in utilizing originating domain is that widgets are forced to connect to a domain when they are installed. The domain can be showed to the user and that information can be used to make the decision about trusting the

widget. A widget can be distributed from a Web portal, with a physical media or from a user to another user. In all distribution cases, the domain linkage always connects the widget to some existing DNS name, which the widget author or distributor must be able to control. Being able to control a domain means in this case that it is possible to control resources uploaded to that domain.

Utilizing originating domain is possible already with an implementation that is done based on the current W3C specification. The requirement is that security policies would be tied to domains rather than certificates. The problem with the current specification is that, as mentioned, widget installation process does not necessarily involve a network connection. The utilization could only be possible when a widget is downloaded directly from the originating domain. The implementation described in this chapter tries to solve this issue and it also brings other benefits that are discussed more in the evaluation chapter of this thesis (chapter 7).

6.2 Installation process

W3C Widgets installation process starts with acquiring the widget resource as depicted in figure 6.1. Once acquired, the resource is validated. The installation process is described in the W3C Widgets 1.0 specification [5] which is coined as traditional installation process in this thesis. In the traditional installation process, the resource validation consists of checking that the widget resource is a valid and suitable Zip-archive and that the MIME type of the resource matches the specified widget MIME type *application/widget*. When the resource validation is performed, the metadata of the widget can be extracted and interpreted. W3C specifies a configuration document where the widget metadata and configuration parameters are defined. A part of the widget metadata is digital signatures [52]. Signatures can be used to authenticate the widget author or distributors and to verify that the widget has not been changed since the signing. W3C widgets can be unsigned or they can include one author signature and/or many distributor signatures.

After the widget user agent has all the necessary information about the widget available, security policies can be taken into account as seen in figure 6.1 on the right hand side. Policies can be simple or complicated and they can originate from many parties as described in section 5.3. Policies might involve user interaction for the trust decision or the policy might be complied just by looking at the widget origin based on the signatures. Policy details and how they are implemented and enforced is outside of the scope of the

W3C Widgets specification. After the policy is programmably complied to or user has decided to trust the widget the installation may proceed. This traditional installation process might be done entirely offline, but it might also require a network connection. For example security policy enforcement might include checking revoked certificates online and making sure that the signer of the widget is still trusted.

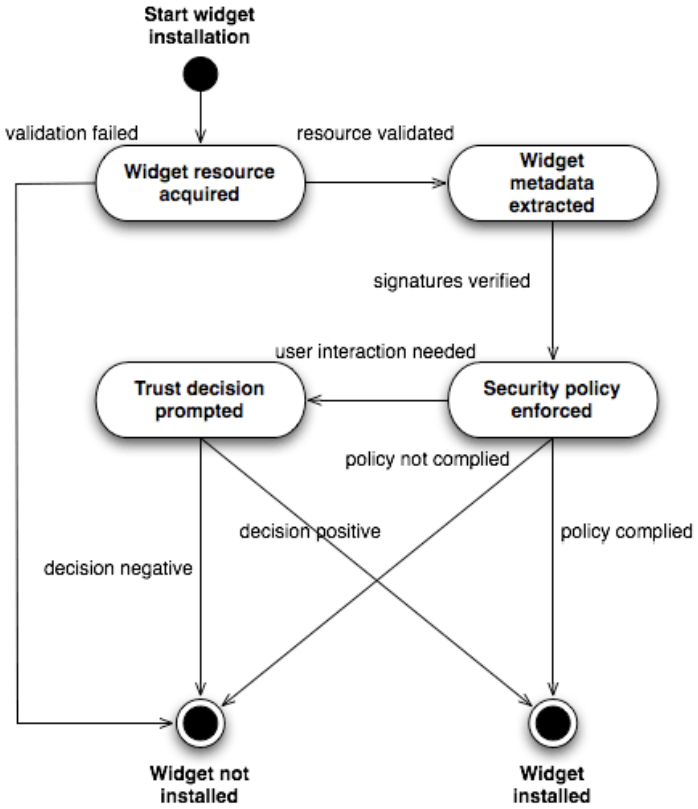

Figure 6.1: Traditional widget installation process

The proposed modification to the traditional widget installation process is presented in figure 6.2. Instead of acquiring a full widget resource, the process begins with acquiring a widget description, which is called a widget stub. The stub contains only the metadata and the configuration parameters which can have an affect on the trust decision about the installation of the widget. An important content of the stub is the information about the full widget resource. Information includes the DNS domain name where the resource can be downloaded and all the digital signatures of the widget. A message digest created by a cryptographic hash function can also included. That can be used to verify the linkage between a stub and a resource. The message

digest can be left optional because the trust is essentially established to an origin and not to a specific content instance. That is why individual widget versions do not need to be recognized and the more important thing is to recognize the origin. Security policy enforcement and user interaction phases are identical to the traditional installation process, but this time there is more information that can be used to make the decision. In addition to the author and distributor signatures the originating domain of the widget is known. Only after a positive trust decision, the actual widget resource is downloaded from the domain defined in the stub. When the resource is downloaded, the defined message digest (if one exists) is used to verify that the downloaded resource matches with the resource described in the stub. If the verification succeeds, the widget can be installed without any user interaction in this phase.

The proposed modification to the widget installation process can be implemented as a *complementary* specification without making any modifications to the original W3C specification or as an *alternative* implementation, which requires more changes to the widget user agent.

6.3 Complementary implementation

In the complementary version, the widget resource can stay unmodified and the widget stub can be used as an alternative way to install it. Resulting widget resource and stub structure of the complementary implementation style can be seen in figure 6.3. The stub contains the configuration document of the widget, the information about the resource and the digital signature files. It does not contain the actual widget implementation files, such as, JavaScript files, images or localization files. Stub content is packaged to a Zip-archive for distribution just like a widget resource content.

A widget user agent, which supports installing from a stub file, first acquires a stub and then extracts the metadata and checks the security policies as described in figure 6.2. The metadata is found from the signature files and from the configuration file. A user agent can also use the configuration file to determine whether the widget is supported by this user agent. The widget might specify some required features in the *feature* element that are not supported or the widget might use only window modes that are not possible in the runtime in question. The *feature* element might also include information about additional APIs that the widget wants to access and that need to conform to the security policy. In error cases, the installation process can be terminated or the user can prompted for a confirmation of a partly

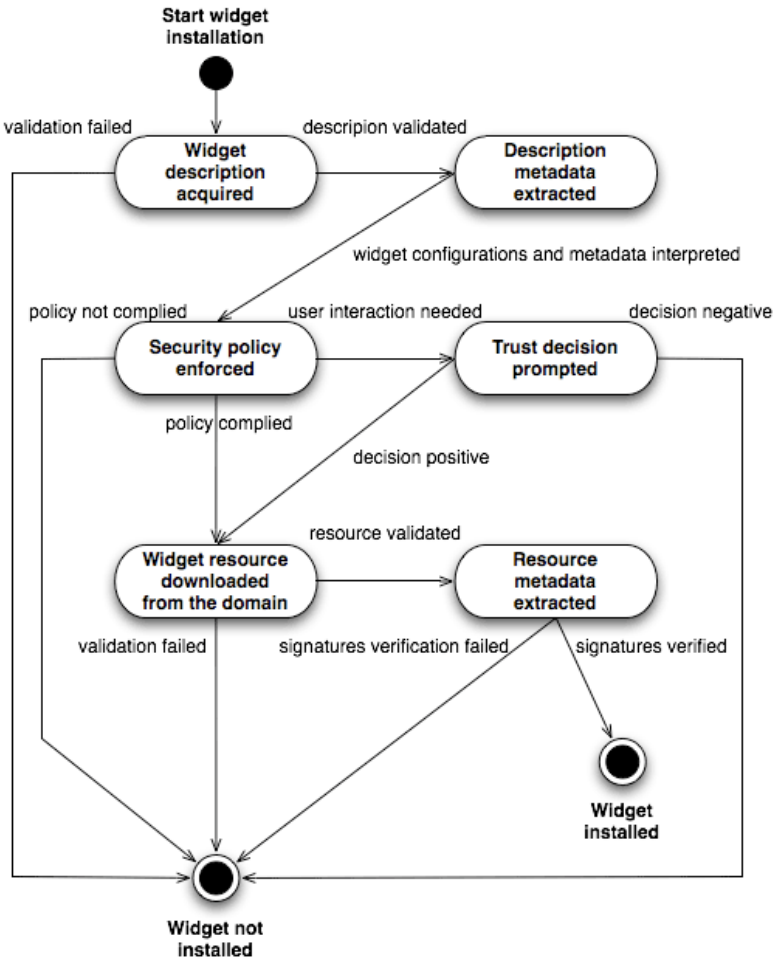

Figure 6.2: Proposed widget installation process that utilizes originating domain

incompatible widget installation or a permission for an API access. In this phase, other descriptive metadata can be shown, such as, the description, the author and the license information of the widget. In regard to the proposed installation process, the most important information shown in this phase is the originating domain. Examples about how this could be visible to the end-user can be found from section 6.6.

The format of the files in the widget resource can be found from the W3C Widgets specification [5]. Widget resource must contain a Configuration Document called *config.xml* and it may contain signature files. Signature files are XML files that have the suffix *.xml* that follow the defined naming convention. Author signature file must be named *author-signature.xml* and

```
resource/
|-- author-signature.xml
|-- config.xml
|-- icon.png
|-- images
|   |-- image1.png
|   `-- image2.png
|-- index.html
|-- scripts
|   |-- script1.js
|   `-- script2.js
`-- signature1.xml

stub/
|-- author-signature.xml
|-- config.xml
|-- resource.xml
`-- signature1.xml
```

Figure 6.3: An example of a widget resource and a widget stub

a resource can contain zero or one author signature. Distributor signature file name is in the format *signature([1-9][0-9]*)?.xml* when expressed as a regular expression. A widget resource may contain any number of distributor signatures. In figure 6.3, the author signature is called *author-signature.xml* and the one and only distributor signature is called *signature1.xml*.

The stub package contains one additional file that is not specified by the W3C. It is the file called *resource.xml*, which contains information about the linked widget resource. The configuration document and the signature files are identical in both packages in the complementary version of the proposed modification. The resource file may contain signatures by the author or distributors, but it can include just the reference to the widget resource. It is important that the signing process is optional because one of the advantages of the proposed modification is the added easiness for the widget author when there is no need to deal with signatures and certificates. In addition to the resource file name a file extension and a MIME type must be defined for the widget stub. The stub describes the widget resource so it is proposed that the MIME type is *application/widget-description* and the file extension *.wdd*.

An example of a resource file without signatures can be seen in figure 6.4. It has a *resource* element as the root element that has a *src* attribute that

points to the location of the widget resource. The format of the resource file is inspired by the format of the Update Description Document (UDD) that is defined in the W3C Widgets Updates specification [62]. One of the design principle of the implementation is to introduce as few new XML elements and constructions as possible. That is why the message digest of the widget resource also reuses definitions from the W3C digital signatures specification. The message digest is described inside a *DigestMethod* element. Only new element, this proposed implementation introduces, is the *resource* element, which should be added to the *widgets* namespace.

The example defines that the used digest algorithm is SHA-256 which is the only algorithm that must be supported to conform the Widgets Digital Signatures specification [52]. SHA-256 can also be considered more secure than SHA-1 or MD5 because they are known to be vulnerable due to found collision mechanisms [63]. Widget user agents may also support additional digest methods. The used method must be defined in the *Algorithm* attribute. In practise, the value of the message digest is calculated from the widget resource file, which is called *widget.wgt* in the example. After calculating the SHA-256 hash from the file, the value is Base64-encoded like the W3C XML Signature specification defines. A reference implementation for calculating the value of the *DigestValue* element written with Python can be found from figure A.2. The resulting value is denoted as three dots which is also the notation used in the W3C digest value examples.

```
<widgets:resource
    xmlns="http://www.w3.org/2000/09/xmldsig#"
    xmlns:widgets="http://www.w3.org/ns/widgets"
    widgets:src="https://example.com/v1.1/widget.wgt">
  <DigestMethod
      Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
  <DigestValue>...</DigestValue>
</widgets:resource>
```

Figure 6.4: An example of a resource file without signatures

As mentioned, it is possible for the author or the distributor to add signatures to the resource file. They can be used to verify that some known party (e.g., party verified by a trusted CA) has defined the source of the widget resource. Signatures to the resource file are proposed to be implemented using the same mechanism that is used in the W3C Widgets Digital Signatures specification. Figure A.3 contains an example of a resource file that contains one distributor signature. Compared to the file without signatures (figure 6.4) this file has

a *Signature* element as a children of the root element. All the elements, attributes and namespaces inside the signature element are defined in the W3C specification. No new elements are introduced. An element called *SignedInfo* is an important element because it defines what information is signed. Here, the signed information is, that this resource file points to a widget resource located in the location which can be found from the *URI* attribute of the *Reference* element. The root *widget* element does not contain a *src* attribute because that information is found from the signed content. Signed is also the digest value of the widget resource so that it can be made sure that the acquired resource was the one that this resource file was about.

The resource file with a distributor signature also contains information about the role of the signer. Roles are defined by the W3C, and in this case the role of the signer is a distributor. An example use case for different roles is that first a company signs an application to prove that it is written by that company. After that a distributor signature is used by another company's network administrator to verify that the application is approved for use within that company. In the end of the file, is the actual signature value and the certificate.

To process a widget stub a user agent must begin with extracting the stub Zip-file. The files inside the stub are processed to find signature files, the Configuration Document and the resource file. Signature files and the Configuration Document are parsed like W3C defines in [5] section 10. That section describes the processing rules for a widget resource package that resemble closely the rules for a widget stub.

The resource file must be called *resource.xml*. Once it is found, it can be examined. To determine whether it contains signatures, the widget user agent must see is there an element called *Signature* as a child of the root element. If there are signatures, they must be processed like defined in [52] section 4, which defines digital signature processing. If there are no signatures, the location of the widget resource can be read directly from the *src* attribute of the root *widget* element. After these steps, the widget user agent has all the information needed for enforcing the defined security policy and finally determining the trust decision. In the proposed widget installation process (figure 6.2) this state is called "Security policy enforced". The resource file could also contain other information about the widget that might affect the user's decision about the installation. One example is the size of the widget resource package. This additional information is loosely related to the trust decision, so it is left outside of the scope of this thesis. However, it is relevant when looking from the user experience perspective.

6.4 Proposed alternative implementation

The complementary implementation above contains redundant data between the widget stub and resource. Same signature files and configuration document must be included in both. The purpose of the proposed alternative implementation is to reduce redundancy, make the installation process more efficient and make the implementation more effortless. The signing implementation is inspired by the implementation that is used to sign Java MIDlet Suite's Java ARchive (JAR) files [64]. MIDlets are Java applications written for the Mobile Information Device Profile (MIDP), which is a profile of Java ME. MIDlet distribution can be done with two files - JAR and Java Application Descriptor (JAD) file. These files can be compared to widget resource and widget stub respectively. JAR file is a Zip-archive that contains the application and a JAD file can be used to hold metadata about the application (e.g., configuration parameters and signatures). According to the MIDP 2.1 specification [65] each JAR file may be accompanied by a JAD file and the JAD file can be used for application management. User agent can determine whether a MIDlet is suitable to run on it based on requirements listed in the JAD file. It also allows to specify configuration-specific attributes that are supplied to the MIDlets without modifying the JAR file (i.e., the application package can stay unmodified). An example JAD file is presented in the figure 6.5. It contains similar information that a widget stub 6.3 does. Widget Configuration Document information can be compared to configuration-specific parameters; JAD can contain signatures from the JAR file and also the location of the JAR is expressed.

```
MIDlet-1: Midletti, icon.png, com.example.midletti.MainMidlet
MIDlet-Name: Midletti
MIDlet-Version: 1.1
MIDlet-Vendor: Example Company
MicroEdition-Profile: MIDP-2.0
MicroEdition-Configuration: CLDC-1.0
MIDlet-Permissions: javax.microedition.io.Connector.sms
MIDlet-Jar-URL: http://example.com/midletti.jar
MIDlet-Jar-Size: 7230
MIDlet-Jar-RSA-SHA1: ...
MIDlet-Certificate-1-1: ...
```

Figure 6.5: An example JAD file that describes an example MIDlet

The syntax of a JAD file is based on key-value pairs separated by a :-

character. W3C does not specify such syntax in the context of JavaScript widgets. The same information can be expressed with XML or JavaScript Object Notation (JSON) [66]. In this thesis, a Widget Description Document (WDD) is proposed to describe widget resources in the same way as a JAD file describes a JAR file. A WDD file must at least contain the location of the widget resource and can contain any number of additional metadata about the widget. Such metadata include signatures or information that is defined in the W3C Widgets specification in the Configuration Document, the update description document or the *access*-element. This information does not need to be repeated in the widget package itself and means that there is no redundant definitions.

WDD file must be recognizable by a widget user agent when encountered from a file system or when downloaded from the Web. That is why WDD files must have a unique file extension and when served over the Internet they must be served with a certain MIME type. In this thesis it is proposed to use *.wdd* as the file extension and *application/widget-description* as the MIME type similarly as in the complementary implementation.

As mentioned, there are two possible syntax selections; either XML or JSON. An example of the same information written with both syntaxes can be found from the figure 6.6. There are many tools that can be used to convert from XML to JSON or vice versa, for example, [1], [2] and [3]. However, if it is decided that both syntaxes are supported, the exact formula of the conversion should be defined formally and in detail. The information presented includes the widget resource location and additional metadata that would be traditionally included in a configuration document. For example, it is defined that the widget needs the camera feature and an application parameter called *apikey* with the preferred value is included. That value must be accessible when the widget is instantiated after a possibly successful installation. This gives the flexibility to change application configuration parameters without changing the actual widget resource package or signatures calculated from that package.

Other than the WDD document, the alternative implementation differs in the way the signing is implemented. The W3C Widgets Digital Signatures specification [52] specifies that the content of the widget package is signed file by file. In more detail, the message digest values of the separate files in the widget package are signed. In this thesis, it is proposed that only the message

[1] http://www.json.org/example.html
[2] http://www.xml.com/pub/a/2006/05/31/converting-between-xml-and-json.html
[3] http://www.xml.lt/Blog/2009/01/21/XML+to+JSON

digest value calculated from the widget resource package is signed. This is a similar approach to the Java MIDlets. In the example JAD (figure 6.5) the signature of the JAR file is the value of the key *MIDlet-Jar-RSA-SHA1*. The public key that corresponds to the used private key is found from the certificate that is the value of the *MIDlet-Certificate-1-1* key. MIDlets can be signed using multiple certificates, but all certificates must be for the same public key. The MIDP 2.1 specification defines: "If multiple CA's are used then all the signer certificates in the application descriptor MUST contain the same public key." [65].

The signature definition format of the WDD is proposed to be different depending on the used markup syntax. When using XML, it is natural to follow the W3C XML Signatures specification. When using JSON, a less verbose format can be used. Figure 6.7 shows how signatures can be described in JSON format. The *author-signature* and *signature1* keys have semantically the same meaning as *author-signature.xml* and *signature1.xml* in figure 6.3. Those were described in more detail in section 6.3. The RSA-SHA256 message digest values are calculated from the widget resource package which is referenced in the example. A reference implementation for calculating the message digest can be found from figure A.2. Certificates are defined as values of a *Certificate-1-1* key. The numbers in the key name are used to include multiple certificates and to express certificate chains. The key name format is *Certificate-<n>-<m>* where n is a number starting from 1 and describing the number of the certificate chain. After that is m which describes the position of a certificate in the chain. This allows signing with intermediate certificates that can be verified from a root certificate after following the chain to the root. Signatures in the WDD file in XML format follow the W3C specification so that the resulting file includes a *Signature* element from which an example can be found from figure A.3.

6.5 Implementation summary

In section 6, two different implementation options were presented that allow the new proposed installation process, which is shown in figure 6.2. The first is called the complementary and the second is called the proposed alternative implementation. The main difference compared to the W3C Widgets [5] is that they both split the widget into two files instead of just the widget resource package. In the complementary implementation, the additional file is called the widget stub and it is a Zip-archive just like the widget resource. In the alternative implementation the additional file is called the Widget

Description Document (WDD), which is not a Zip-archive, but a textual file. In both of the implementations the purpose of the splitting is to allow the usage of the originating domain to make a trust decision. Splitting brings also other benefits that are discussed in the section 7.

The main difference between the complementary implementation and the proposed alternative is the way how signatures are defined. The complementary follows more closely to the W3C Widgets Digital Signatures [52] specification and signed content is individual files inside a widget resource package. Alternative implementation takes more properties from the Java MIDlet signature implementation where the package is signed, and not the individual files. Another difference is that the alternative implementation can be made more efficient by including some or all of the metadata about the widget to the WDD and removing it from the widget resource package. However, this breaks the compatibility with the current W3C widgets specification. After removing the metadata from the widget package, it cannot be installed anymore to a user agent that only supports the W3C-defined installation process.

The alternative implementation includes two options for the syntax of the WDD file - XML and JSON. When using XML, W3C XML Signatures [43] can be used and the widget metadata can be expressed in the XML file like defined by the W3C. When using JSON, the signature format can be made simpler and a mapping from the metadata configurations from XML to JSON is needed.

6.6 User interface examples

The proposed implementations allow the usage of the originating domain as the source of the trust. This section contains User Interface (UI) examples, which demonstrate how the implementations could be visible to the user. In figure 6.8, the prompt on the left side would be shown to the user if the widget comes from an arbitrary domain, does not use secure connection such as SSL/TLS and does not contain signatures. To help the user to make the trust decision, the originating domain is the biggest textual element in the prompt. The red color is used to mark that the platform was unable to verify the origin and that the user should notice the domain. The next prompt in the figure could be shown if the widget from an unverified source uses some privileged APIs. In the example, the widget needs access to the location, network and camera. User can either allow or disallow the access. After a positive trust decision, the installation may begin and the actual

widget resource will be downloaded. No user interaction is needed after this phase, so the user can perform other tasks while the widget is downloaded and installed. This can be shown to the user, for example, with the third prompt in the figure.

If the widget user agent has more information about the trustworthiness of the source it must be visible to the user. In figure 6.9, the leftmost prompt could be used if the widget is from an arbitrary domain and uses secure connection with a certificate issued by a trusted CA. The green color of the domain name is used to indicate that the source can be more likely trusted than in the first example where red color was used. Yellow color is used in the prompt in the middle when the widget uses secure connection, but the certificate can not be verified. In that situation the integrity of the downloaded widget is guaranteed but the source can not be authenticated by the user agent. The third prompt in the figure shows a green company name instead of a domain name. This kind of prompt could be used when signatures are used and the certificate can be verified. When the user agent can authenticate the source, there might be no need for an access control prompt like in the middle of the figure 6.8. Security policies can be defined so that widgets from a certain origin or signed by certain parties are automatically granted the access to some or all privileged APIs.

```
<?xml version="1.0" encoding="UTF-8"?>
<widget xmlns="http://www.w3.org/ns/widgets"
        src="https://example.com/v1.1/widget.wgt"
        viewmodes="application fullscreen">
  <name>
    The example Widget
  </name>
  <feature name="http://example.com/camera">
    <param name="autofocus" value="true"/>
  </feature>
  <preference name="apikey"
              value="ea31ad3a23fd2f"
              readonly="true"/>
</widget>
```

```
{
    "widget" : {
        "src" : "https:\/\/example.com\/v1.1\/widget.wgt",
        "viewmodes" : "application fullscreen",
        "name" : "The example Widget",
        "feature" : {
            "name" : "http:\/\/example.com\/camera",
            "param" : {
                "name" : "autofocus",
                "value" : true
            }
        },
        "preference" : {
            "name" : "apikey",
            "value" : "ea31ad3a23fd2f",
            "readonly" : true
        }
    }
}
```

Figure 6.6: Widget description file with XML and JSON syntax

```
{
  "widget" : {
      "src" : "https:\/\/widget.example.com\/v1.1\/widget.wgt",
      "author-signature" : {
          "RSA-SHA256" : "...",
          "Certificate-1-1" : "..."
      },
      "signature1" : {
          "RSA-SHA256" : "...",
          "Certificate-1-1" : "..."
      }
  }
}
```

Figure 6.7: Widget description file signature definition format when using JSON syntax

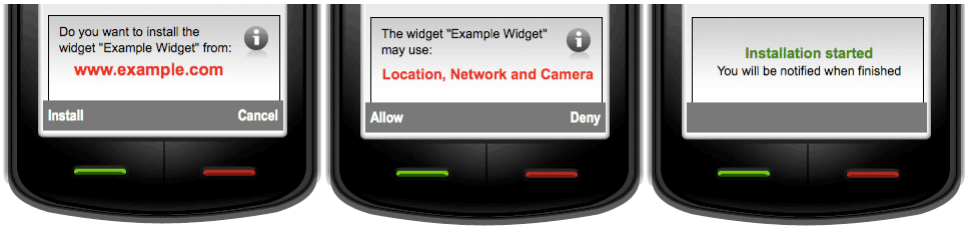

Figure 6.8: Example security prompts for a widget from an arbitrary domain

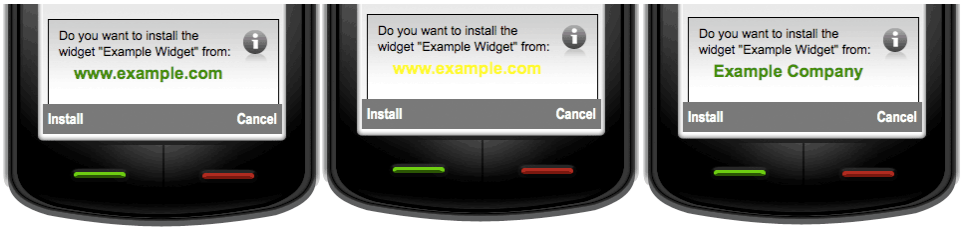

Figure 6.9: Example prompts for widgets with various source configurations

Chapter 7

Evaluation

In this chapter, the implementation described in the previous chapter is analyzed based on the design requirements from chapter 5. Also the benefits and drawbacks of the two implementation styles and other existing solutions are presented. The trust establishment implementation covers only the widget installation process. Restricting the capabilities that are granted during the installation is discussed in the last section of this chapter.

7.1 Evaluation against design requirements

This section contains an evaluation of the implementation described in chapter 6 based on the trust establishment requirements described in section 5.5. The list of requirements is gone through item by item and it is evaluated how well does the implementation meet the requirements?

Effortless mapping to policies Mapping can be done by listing DNS names and associating the policy and granted privileges to each origin. DNS names are human-readable so a policy definition list can be interpreted by looking at the policy file and it can be understood without additional documentation. Mapping from DNS names to capabilities allows hierarchical and flexible mappings. For example, a regular expression `https://[^.]*.example.com` could be used to match all domains which are directly under `example.com` and are protected with a secure connection.

User experience A user that is used to browse the Web has at least some conception of DNS names and what they are about.

Developer experience DNS names have a relatively long history in the Internet era so they are well known. It is easy to acquire a DNS name and most developers and companies already have one. The cost for a DNS domain name is low. There is no need for separate development time certifications (i.e., developer certificates) if the application can be downloaded from a trusted origin during the development. Name-based policies and implementations are easier to debug than cryptographic hashes and signatures.

Application updatebility When the trust is based on the originating domain, updates can be deployed without renewing the established trust. Application author can upload the new application to the server and the existing trust relation and related policies will remain.

Flexibility The content itself is not examined when the trust is based on the origin. It brings flexibility to the development and deployment. When the application package and the description file are separated it is possible to configure the application and the related metadata without modifying the application.

Distributor control Distributors can define their own policies and define which of the origins are trusted. When the application description is separated, a distributor can sign content as trusted without having the control of the widget package.

Implementation effort Most platforms that will support JavaScript widgets probably already have a Web browser implemented. It means that most required components already exist. Those include used protocols, such as, DHCP, SSL/TLS and HTTP.

Easy to understand Because of the human-readable nature of DNS domain names, they are relatively easy to understand when comparing to digital signatures and certificates.

Compatibility with the existing infrastructure The proposed implementation does not introduce new technologies as such and can be realized with the existing infrastructure. DNS domain name system is so fundamental mechanism of the Web that it is probably not going to change or disappear rapidly.

7.2 Comparisons

7.2.1 Certificates versus originating domain

Costs Both certificates and DNS domain create costs. As an example at 1.8.2009 the costs for a *.com* domain were starting from 5 euros per year and for a code signing certificate 140 euros per year from an example service provider called Go Daddy [1]. DNS name require a DNS server configuration that adds some costs. DNS name can be protected with a SSL/TLS encryption to authenticate the server and to guarantee integrity, and the costs for that at Go Daddy start from around 20 euros per year. Usually, companies and developers already have a DNS domain acquired so in those cases this does not create any additional costs.

Security Certificates can be considered a more secure indicator of the origin. The used cryptography does not contain any significant weaknesses and the authentication of the real-world entity is done more reliably than when acquiring a non-secure *.com* domain for example. However, if encryption is used, the authentication process is similar in both. DNS contains some known vulnerabilities like Denial of Service (DoS) and DNS rebinding [26]. They are threats to the domain-based origin model, but are difficult enough to carry out in practise so that they do not jeopardize the applicability of the model. Centralizing the trust authority creates a big risk if the authority is compromised. For example, if a private key of a trusted CA leaks, it is possible to create fake certificates that look like verified by the CA.

Authoring Certificates are used together with digital signatures that are calculated on per widget instance basis. It means that every time the widget changes a new signature process must be carried out. The domain-based model uses the origin of the code as the trust indicator so different widget versions are considered equally trusted as long as they come from the same source. In the JavaScript widget environment, the party responsible for the code is more important than the code itself. This is because widgets can be dynamic and they can change themselves after they have been installed (section 7.5).

User experience If a widget author does not have the opportunity to use certificates and is forced to deploy a widget without signatures or with a

[1] http://www.godaddy.com

self-signed certificate there is very little information that can be showed to the user when making a trust decision. A DNS domain offers at least some hint to the user about the origin of the application even if the origin is not verified by a trusted party. Another view point in regards to the user experience is that domain names are known to the user at least on some level. They are used in other trust-related decision in the context of W3C Widgets. The Widget Access Request Policy [53] specifies how to define the network access policy of a widget. The policy definition includes a list of domains that are accessible by the widget. It means that if these access policy definitions are used, a widget user agent is anyway forced to determine the trust towards a domain. The policy conformation might require user interaction, which means that user has to think about DNS domains.

Policy reusability There are use cases where domain names are used as the keys when defining security policies. An example is the Geolocation API and how it is implemented in Firefox [2]. Users can grant Web sites a permission to access the location information. It means that domain names are mapped to access control policies. Same policies could be used between a browser and a widget user agent. It is expected that in the future more and more APIs will be made available from the widget and also from the browser execution context [59].

Monopolism In a situation where a certification from a certain Certificate Authority is needed to run an application on a runtime, it can be said that the CA has a monopolistic position. All application authors must purchase a certification service to be able to deploy on that particular runtime. Leveraging DNS mechanism does not create such monopolism that easily because the DNS infrastructure is not controlled by an individual company. It is a world-wide system with a hierarchical structure 2.4.

Past experiences Signatures and certificates are the trust mechanism for J2ME MIDlets. Today, the related ecosystem is very complex and does not work well. It is said that the signing is even killing the whole technology because it makes application development and deployment so hard [3] [4] [5]. The costs of the required certificates have become unbear-

[2] http://www.mozilla.com/en-US/firefox/geolocation/
[3] http://javablog.co.uk/2007/08/09/how-midlet-signing-is-killing-j2me/
[4] http://www.spenceruresk.com/2007/05/26/the-hidden-problem-with-j2me/
[5] http://blog.javia.org/midlet-signing/

able for individual developers. The reasons for the failure are not so much technical though, but rather business and political reasons.

7.2.2 One package versus two packages

Flexibility A third party can sign a widget as trusted without the widget author even knowing about it. As an example an operator could sign a widget that is distributed by a third-party developer so that the actual widget package does not need to be touched. The signatures are added to the widget stub or to the Widget Description Document.

User experience A widget stub or a WDD is smaller than the widget package, which means that it can be acquired faster. In mobile context, when the used network connection is slow, the time difference can be remarkable. Faster acquiring means that the user can make the trust decision and the decision about installing a widget more instantly. Only after a positive decision the widget package is downloaded. Widget compatibility with the user agent in question can also be checked based on the first file. If only one widget package would be used it should first be downloaded as whole and in a case of a negative decision there would be unnecessary download rate used. This might create costs to the user if flat rate data plan is not used. In the installation experience flow, it is more natural to make all decisions as close to each other as possible. The first decision is to download a widget (e.g., click a download link from a widget portal) and the second is whether to actually install the downloaded widget. If the widget package download takes, for example, 20 minutes the user flow has already been broken if the installation decision has to be made only after the download.

Distribution If a widget is separated into two packages it can be distributed more easily. The actual widget package can be hosted in one place where it is easy to manage and update and the descriptor file can be the one distributed. For example, a company can have a widget catalogue hosted at their domain, but WDD files of the widgets can be uploaded to several widget repositories and portals.

7.2.3 Complementary versus alternative implementation

Signing individual files The alternative implementation does not support signing individual files. The widget package altogether is signed. There

is usually no need to sign files separately. It complicates the signing process and the trust decision is a binary choice about whether to install an application or not. Individual source code files are not good separately and they always are an integral part of a total widget.

Metadata syntax Complementary implementation uses XML to describe metadata, but alternative implementation includes an optional use of JSON. JSON creates smaller files because it is less verbose than XML. Parsing JSON is also becoming faster because native JSON parsing has been implemented in some of the latest popular browsers and widget user agents are often based on those browser engines. However, XML can be also made more efficient with binary formats, such as, the *Xebu* [67].

XML Signatures The complementary method uses XML Signatures which are more complex and harder to implement than the signature method in the alternative implementation. W3C Widgets Digital Signatures require XML to be canonical and because of that, the widget user agent must also implement at least the one required canonicalization algorithm [52].

7.3 Evaluation against browser security model limitations

JavaScript security model limitations in browsers were discussed in section 3.1. The SOP-based security model that relies heavily on DNS domains was described to contain many limitations and weaknesses. Browser security model is related to widget user agents because the underlying technology stack is the same in both. Widgets also run on top of engines that are capable of executing applications written with Web technologies. In this section, the list of browser security model limitations is evaluated in the context of W3C Widgets and the proposed trust establishment mechanism.

SOP workarounds W3C Widgets Access Requests Policy specification deals with widget network access control. It is possible to make a whitelist of domains that are accessible from the widget execution context. This helps to deal with SOP workarounds, but requires that the access policy also affects to *script*- and *img*-tags in addition to the XHR access.

Cross Site Scripting (XSS) XSS attacks cannot be coped with the Access Requests Policy, but the damage that the attack causes can be reduced. Even if an attacker can inject JavaScript code to an execution context, it is not possible to steal data that easily when the access outside the context is managed with policies and it is not possible to make a connection to an arbitrary domain. The risk for an XSS attack still remains if proper input/output validation is not done properly or there is no other mechanism in place that prevents them.

Cross Site Request Forgery (CSRF) The damage of a CSRF is usually considerable if HTTP cookies are used for identifying and authenticating the client. Widgets can have more novel ways of storing state information in the client-side. For example, in W3C there is the Web Storage specification [68] that can be used to store data in the client-side if the specification is implemented by the widget user agent.

Session stealing Session stealing is as big of a problem with widgets than it is in the traditional browser environment. The main reason is that the communication channel between a client and a server can be eavesdropped if it is not secured. However, some widgets might not need a server-side component and they could run only locally. The proposed implementation forces a connection to a domain. If session information is used during the connection it can be said that the proposed implementation is more vulnerable to session stealing. A game widget without the need to connect anywhere is safe because there is no session to steal.

Absent content security methods This limitation is even more risky with widgets because many widget runtimes offer privileged APIs that can be called from widgets, but not from a browser environment. W3C has formed a Device APIs and Policy Working Group Charter [6] who's task is to create a framework and policies for defining access control for security-critical APIs. An access control policy is needed for a content and API access so that they are harder to abuse by malicious parties.

Implicit trust in domains The proposed implementation makes the trust towards a domain more explicit. The domain is trusted either by user's direct decision or by the fact that a pre-defined policy indicates that the domain is trusted. Also the Access Requests Policy deals with this limitation because the widget author has the possibility to explicitly list which domain can be connected to (i.e., can be trusted).

[6] http://www.w3.org/2009/05/DeviceAPICharter

Integrity issues A widget runtime has the same technologies available for integrity issues than a browser runtime. SSL/TLS can be used for transport layer integrity and there exists also technologies and methods for more higher level integrity. Digital signatures defined in the Widgets specification are one of the enablers that can be used for application level integrity. They can be used in the traditional W3C-defined implementation and in the proposed implementation.

7.4 Other possible solutions

7.4.1 Decentralized trust

The trust model is centralized with certificates and the PKI. The root of the trust is CAs that certify other parties. An alternative model is to decentralize the trust. One example concept that leverages this model is the *web of trust* [69]. It is used in Pretty Good Privacy (PGP) to bind public keys to users. In PKI, public keys are bind to real-world entities by CAs, but in web of trust binding is done by other users of the system. The same kind of concept could be used to make a trust decision about an application. If users that are in your web of trust have trusted an application it could be trusted by you too.

Similar content trustworthiness concepts have been developed for Peer-to-Peer (P2P) networks where users share resources with each other. An example of such an approach is the *EigenTrust* algorithm [70]. It is a reputation system where a trust value is calculated for each peer in the network. The value is used to determine from whom other peers download their files and to isolate malicious peers from the network. Same kind of reputation system has been proposed at [56]. Reputation models could be adapted to application trust establishment by calculating trust values to application authors, distributors or applications themselves. Values could be based on a feedback or rankings given by other users or recommendations made by other developers.

7.4.2 Source code and behaviour analysis

One approach to determine whether an application can be trusted is to look at the application itself. Other approaches proposed or presented in this thesis rather look at where the application comes from or who has verified that the application can be trusted. Source code could be automatically

analyzed in two ways - either statically or dynamically. In statical source code analysis, the application is not executed. The source is interpreted by an external computer program that tries to determine properties about the source. In dynamic analysis, the source code is executed and the behaviour is examined during the run time.

In the case of JavaScript widgets it could be determined that the application does not call any sensitive APIs and that it does not use a network connection at all. Those applications could be installed with relaxed security policies (e.g., by not prompting a permission from the user). Statically this could be done, for example, by using regular expressions to look for certain patterns or API calls from the source [29].

Dynamic analysis could be done by monitoring JavaScript applications and evaluating the actions that the application performs. Auditing information and intrusion detection techniques can be used to detect malicious actions [71]. Another dynamic behaviour detection mechanism is to examine the HTTP traffic that a JavaScript application causes [30].

7.5 Restricting granted capabilities

After the security policy for widget installation is enforced, widget is installed and the widget has been given a set of capabilities. The capabilities must be restricted so that they are granted only to the correct application or even only to parts of it. Capabilities are things like the capability to install the application, to update it, to access the network or to access some private information about the user. In a mobile device an example is a capability to read user's SMS messages. A more cross-platform example is the location information. W3C specifies the Geolocation API [72] that is implemented, for example, in the latest Firefox version [7] and in the iPhone version of the Safari browser [8]. It allows JavaScript code to access user's location through an API call.

In the case of JavaScript applications restricting capabilities is harder than with some other programming languages. The security model has limitations like described in section 3.1 and the dynamic nature of the language makes it possible for applications to change their behaviour. For example, a signed JavaScript widget might become intentionally or unintentionally malicious after installation through a XSS attack or after an update. Applications

[7] http://www.mozilla.com/en-US/firefox/geolocation/

[8] http://blogs.computerworld.com/iphones_safari_browser_to_include_geolocation_0

might also mashup data from different sources, so they might be vulnerable if some of the sources are compromised or starts to behave maliciously.

There exists lots of research around JavaScript sandboxing and secure mashups in browser environments like discussed in chapter 3. Although many of them are primarily designed for embedding JavaScript programs from different sources to a single Web page, some of them are applicable for restricting granted capabilities in the context of W3C Widgets. An important difference between browser mashups and widgets is that widgets are client-side stand-alone applications. Browser mashups are more like widget containers that can include many widgets running in a single browser window and that require some kind of server component.

An inter-widget communication protocol has been proposed [73] and it is referred in the W3C Widgets URI Scheme [74] specification, but it is not yet specified by W3C. For this reason, in this thesis widgets are considered as independent applications and the restriction of capabilities is considered only in the scope of a single widget.

One option to restrict capabilities is to sandbox JavaScript code like done with signed scripts (section 3.3). Scripts that are signed may be granted extra capabilities, but they cannot interact at all with unsigned code. This can be seen as a full restriction and provides good security. The downside of the method is that it requires support from the widget engine. It cannot be leveraged as a cross-platform solution because there is no common signed scripts support available on different platforms. The method also limits some use cases when the code needs to be able to update itself or when it needs to communicate with unsigned code.

The next level of restriction can be done on the JavaScript level rather than in the platform. Examples are Google Caja [75], FBJS [76] and Microsoft WebSandbox [77]. They are ways to "sanitize" JavaScript code by doing a source-to-source translation. JavaScript code is translated to a format that allows better control of the running code. In Caja, the target is to allow object-capability programming model. Without translations, object-capability model is hard to implement because JavaScript has a global variable that is accessible by all JavaScript code. It means that there is no way to protect modules or parts of the application. When a malicious script gets to run in a JavaScript context, it has access to all information that the other code has too (i.e., granted capabilities also leak to the malicious code). Caja, FBJS and WebSandbox define a subset/superset of JavaScript and prevent developers to use the standardized language. They also cause a performance penalty and result in bigger applications. This is because additional logic

and run-time checks have to be added to achieve the pursued goal. For example the authors of the WebSandbox tell in their presentation that the performance is 1.5-4 times lower when using their implementation [78].

Besides source-to-source translation, JavaScript language can be made more secure by using a safer subset from the beginning. This is the approach taken in ADsafe [79]. ADsafe defines a safe subset of the JavaScript language so that applications that use only that subset can be safely embedded into a Web page. The primary use case is advertising. Third party advertisements coded with JavaScript are safe to embed if they are coded using the ADsafe subset. The conformance can be checked with a validator. The advantage over translating approaches is that no additional code is injected to the application, so it can run efficiently.

Adobe AIR introduces a security sandbox solution for restricting privileged API access to only to the initial application code. Initial application code is code that is packaged in the application and that code runs in an *application sandbox*. Code running in an application sandbox has access to AIR APIs such as the file system API. It is not possible to load code outside the application package to the sandbox. Files and code loaded outside the application are placed to a *non-application sandbox* which has more relaxed policy. Code running there can load scripts from the network and create JavaScript code dynamically. For example the use of *eval* is prohibited in the application sandbox, but can be used in the non-application environment. Non-application sandbox prevents accessing APIs that could potentially be used for malicious purposes. AIR also defines a bridge API that can be used to communicate between the different sandboxes. [51]

All the methods above can be used to restrict granted capabilities. Most importantly, they protect against attacks such as XSS and CSRF. They do not help in the case where the widget author intentionally develops a malicious widget. In those cases, there should be mechanisms to block the access from widgets that are noticed to behave maliciously. With certificates, a certificate revocation mechanism could be used and in case of domain-based trust model a dynamic blacklisting of domains could be in place.

Chapter 8

Conclusion

In this thesis, a new trust establishment mechanism for installing JavaScript applications was implemented on top the W3C Widgets specification. Instead of digital signatures the new mechanism is built on DNS domains. DNS is a fundamental system in the Web infrastructure and the way trust authority is hierarchically distributed in it suits well with the JavaScript widgets ecosystem. In the DNS based model, the trust is formed towards an origin of an applications rather than towards a single application instance. This brings flexibility to the application development and distribution phases. The new trust establishment mechanism makes it possible to create more user-friendly installation flow where the trust decision can be made before acquiring the actual application package.

In addition to the implementation details, the implementation was evaluated against a set of defined design requirements. The evaluation indicated that the new proposed trust establishment mechanism has potential. Digital signatures have failed to fulfil their purpose in the case of Java MIDlets. DNS based model has properties that could prevent such a failure. One of them is the fact that the DNS is not controlled by companies and it is a distributed world-wide system. This could help to keep the application ecosystem open for individual developers and small companies.

The research question was: How can trust be established to provide usable security? The new mechanism provides sufficient security and brings usability improvements from which multiple parties can benefit. These parties include application authors, distributors and end-users.

8.1 Future work

In this thesis, a new trust establishment mechanism was proposed. By using the mechanism it is possible to implement an application installation flow that is more user-friendly. One future work item is to measure the impact of the new flow and compare it to the existing solutions using user experience measurement techniques.

Another area for future work is to study how the development of the ECMAScript standard affects solutions discussed in this thesis. New versions of the specification are being developed which address the security problems with JavaScript today.

Bibliography

[1] Z. Yan and S. Holtmanns, "Trust modeling and management: from social trust to digital trust," *Idea Group Inc.*, 2007.

[2] W. A. S. Consortium. (2009) The web hacking incidents database. [Online]. Available: http://www.xiom.com/whid

[3] The World Wide Web Consortium. (2007) Widgets 1.0: Requirements. [Online]. Available: http://www.w3.org/TR/2007/WD-widgets-reqs-20070209/

[4] I. Strategy Analytics. (2008) Understanding the mobile ecosystem. [Online]. Available: http://www.adobe.com/devnet/devices/articles/mobile_ecosystem.pdf

[5] The World Wide Web Consortium. (2009) Widgets 1.0: Packaging and configuration. [Online]. Available: http://www.w3.org/TR/2009/CR-widgets-20090723/

[6] N. Ben-Asher, J. Meyer, S. Moller, and R. Englert, "An experimental system for studying the tradeoff between usability and security," *Availability, Reliability and Security, International Conference on*, vol. 0, pp. 882–887, 2009.

[7] E. International. (1996) Ecmascript language specification, standard ecma-262, 3rd edition. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[8] D. J. Bouvier, "Versions and standards of html," *SIGAPP Appl. Comput. Rev.*, vol. 3, no. 2, pp. 9–15, 1995.

[9] M. Corporation. (2009) About dynamic content. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms533040(VS.85).aspx

[10] The World Wide Web Consortium. (1998) Document object model (dom) level 1 specification. [Online]. Available: http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/

[11] ——. (1996) Cascading style sheets, level 1. [Online]. Available: http://www.w3.org/TR/CSS1/

[12] D. E. Comer, *Internetworking with TCP/IP Vol.1: Principles, Protocols, and Architecture (4th Edition).* Prentice Hall, 2000.

[13] J. J. Garrett. (2005) Ajax: A new approach to web applications. [Online]. Available: http://www.adaptivepath.com/ideas/essays/archives/000385.php

[14] The World Wide Web Consortium. (2008) The XMLHttpRequest Object. [Online]. Available: http://www.w3.org/TR/XMLHttpRequest/

[15] B. Wilson. (2008) Mama: Scripting - quantities and sizes. [Online]. Available: http://dev.opera.com/articles/view/mama-scripting-quantities-and-sizes/

[16] T. Powell and F. Schneider, *JavaScript: The Complete Reference.* McGraw-Hill/Osborne Media, 2004.

[17] J. Ruderman. (2008) Same origin policy for javascript. [Online]. Available: https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript

[18] M. Zalewski. (2008) Browser security handbook. [Online]. Available: http://code.google.com/p/browsersec/

[19] A. Barth, C. Jackson, and J. C. Mitchell, "Securing frame communication in browsers," in *In Proc. of the 15th ACM Conf. on Computer and Communications Security (CCS 2008).* USENIX Association, 2008, pp. 17–30.

[20] V. Anupam and A. Mayer, "Security of web browser scripting languages: vulnerabilities, attacks, and remedies," in *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium.* Berkeley, CA, USA: USENIX Association, 1998, pp. 15–15.

[21] D. Yu, A. Chander, N. Islam, and I. Serikov, "Javascript instrumentation for browser security," in *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* New York, NY, USA: ACM, 2007, pp. 237–249.

[22] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "Browsershield: Vulnerability-driven filtering of dynamic html," *ACM Trans. Web*, vol. 1, no. 3, p. 11, 2007.

[23] S. Crites, F. Hsu, and H. Chen, "Omash: enabling secure web mashups via object abstractions," in *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security.* New York, NY, USA: ACM, 2008, pp. 99–108.

[24] H. J. Wang, X. Fan, J. Howell, and C. Jackson, "Protection and communication abstractions for web browsers in mashupos," in *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles.* New York, NY, USA: ACM, 2007, pp. 1–16.

[25] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner, "Dynamic pharming attacks and locked same-origin policies for web browsers," in *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security.* New York, NY, USA: ACM, 2007, pp. 58–71.

[26] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, "Protecting browsers from dns rebinding attacks," in *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security.* New York, NY, USA: ACM, 2007, pp. 421–431.

[27] OWASP. (2007) Top 10 2007 - owasp. [Online]. Available: http://www.owasp.org/index.php/Top_10_2007

[28] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *WWW '07: Proceedings of the 16th international conference on World Wide Web.* New York, NY, USA: ACM, 2007, pp. 601–610.

[29] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *ICSE '08: Proceedings of the 30th international conference on Software engineering.* New York, NY, USA: ACM, 2008, pp. 171–180.

[30] B. Engelmann, "Dynamic web application analysis for cross site scripting detection," Master's thesis, University of Hamburg, 2007.

[31] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security.* New York, NY, USA: ACM, 2008, pp. 75–88.

[32] A. van Kesteren. (2008) Access control for cross-site requests - w3c working draft 12 september 2008. [Online]. Available: http://www.w3.org/TR/2008/WD-access-control-20080912/

[33] W. Zeller and E. W. Felten. (2008) Cross-site request forgeries: Exploitation and prevention. [Online]. Available: http://www.freedom-to-tinker.com/sites/default/files/csrf.pdf

[34] A. Liu, J. Kovacs, C.-T. Huang, and M. Gouda, "A secure cookie protocol," *Computer Communications and Networks, 2005. ICCCN 2005. Proceedings. 14th International Conference on*, pp. 333–338, Oct. 2005.

[35] H. Xia and J. C. Brustoloni, "Hardening web browsers against man-in-the-middle and eavesdropping attacks," in *WWW '05: Proceedings of the 14th international conference on World Wide Web*. New York, NY, USA: ACM, 2005, pp. 489–498.

[36] I. N. W. Group. (2008) The transport layer security (tls) protocol version 1.2. [Online]. Available: http://tools.ietf.org/html/rfc5246

[37] M. Myers, VeriSign, R. Ankney, et al. (1999) X.509 internet public key infrastructure online certificate status protocol - ocsp. [Online]. Available: http://tools.ietf.org/html/rfc2560

[38] O. M. A. Ltd. (2007) Oma online certificate status protocol mobile profile v1.0 approved enabler. [Online]. Available: http://www.openmobilealliance.org/technical/release_program/ocsp_v10_a.aspx

[39] N. C. Corporation. (1997) Netscape object signing: Establishing trust for downloaded software. [Online]. Available: http://docs.sun.com/source/816-6171-10/owp.htm

[40] J. Ruderman. (2007) Signed scripts in mozilla. [Online]. Available: http://www.mozilla.org/projects/security/components/signed-scripts.html

[41] Microsoft. (2007) How to use security zones in internet explorer. [Online]. Available: http://support.microsoft.com/kb/174360

[42] V. Gupta, R. Franco, and V. Kudulur. (2005) Dude, where's my intranet zone? (...and more about the changes to ie7 security zones). [Online]. Available: http://blogs.msdn.com/ie/archive/2005/12/07/501075.aspx

[43] The World Wide Web Consortium. (2008) Xml signature syntax and processing (second edition). [Online]. Available: http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/

[44] B. Sterne. (2009) Content security policy. [Online]. Available: http://people.mozilla.org/~bsterne/content-security-policy/index.html

[45] G. Markham. (2007) Content restrictions. [Online]. Available: http://www.gerv.net/security/content-restrictions/

[46] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji, "Soma: mutual approval for included content in web pages," in *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2008, pp. 89–98.

[47] A. S. Incorporated. (2009) Cross-domain policy file specification. [Online]. Available: http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html

[48] The World Wide Web Consortium. (2009) Cross-origin resource sharing. [Online]. Available: http://www.w3.org/TR/2009/WD-cors-20090317/

[49] OpenSocial Foundation. (2009) Opensocial. [Online]. Available: http://www.opensocial.org/

[50] Apple. (2007) Developing dashboard widgets. [Online]. Available: http://developer.apple.com/macosx/Dashboard.html

[51] Adobe. (2007) Adobe air security white paper. [Online]. Available: http://download.macromedia.com/pub/labs/air/air_security.pdf

[52] The World Wide Web Consortium. (2009) Widgets 1.0: Digital signatures. [Online]. Available: http://www.w3.org/TR/2009/WD-widgets-digsig-20090430/

[53] ——. (2009) Widgets 1.0: Access requests policy. [Online]. Available: http://www.w3.org/TR/2009/WD-widgets-access-20090618/

[54] J. Linn. (2000) Trust models and management in public-key infrastructures. [Online]. Available: ftp://ftp.rsasecurity.com/pub/pdfs/PKIPaper.pdf

[55] Z. Yan, V. Niemi, Y. Dong, and G. Yu, "A user behavior based trust model for mobile applications," in *ATC '08: Proceedings of the 5th international conference on Autonomic and Trusted Computing*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 455–469.

[56] Y. Wang and J. Vassileva, "Trust and reputation model in peer-to-peer networks," in *P2P '03: Proceedings of the 3rd International Conference on Peer-to-Peer Computing.* Washington, DC, USA: IEEE Computer Society, 2003, p. 150.

[57] B. J. Fogg, C. Soohoo, D. R. Danielson, L. Marable, J. Stanford, and E. R. Tauber, "How do users evaluate the credibility of web sites?: a study with over 2,500 participants," in *DUX '03: Proceedings of the 2003 conference on Designing for user experiences.* New York, NY, USA: ACM, 2003, pp. 1–15.

[58] The World Wide Web Consortium. (2008) Widgets 1.0: The widget landscape (q1 2008). [Online]. Available: http://www.w3.org/TR/2008/WD-widgets-land-20080414/

[59] ——. (2008) Security for access to device apis from the web - w3c workshop. [Online]. Available: http://www.w3.org/2008/security-ws/report

[60] B. Christianson and W. S. Harbison, "Why isn't trust transitive?" in *Proceedings of the International Workshop on Security Protocols.* London, UK: Springer-Verlag, 1997, pp. 171–176.

[61] Apple. (2009) App store and applications for iphone. [Online]. Available: http://www.apple.com/iphone/appstore/

[62] The World Wide Web Consortium. (2009) Widgets 1.0: Updates. [Online]. Available: http://www.w3.org/TR/2008/WD-widgets-updates-20081007/

[63] X. Wang, Y. L. Yin, and H. Yu, "Finding collisions in the full sha-1," in *In Proceedings of Crypto.* Springer, 2005, pp. 17–36.

[64] J. Knudsen. (2003) Understanding midp 2.0's security architecture. [Online]. Available: http://developers.sun.com/mobility/midp/articles/permissions/

[65] J. . E. Group. (2006) Mobile information device profile for java 2 micro edition version 2.1. [Online]. Available: http://www.jcp.org/en/jsr/detail?id=118

[66] D. Crockford. (2006) The application/json media type for javascript object notation (json). [Online]. Available: http://tools.ietf.org/html/rfc4627

[67] J. Kangasharju, T. Lindholm, and S. Tarkoma, "Xml messaging for mobile devices: From requirements to implementation," *Comput. Netw.*, vol. 51, no. 16, pp. 4634–4654, 2007.

[68] The World Wide Web Consortium. (2009) Web storage. [Online]. Available: http://www.w3.org/TR/2009/WD-webstorage-20090423/

[69] C. Ellison. (2001) Spki/sdsi and the web of trust. [Online]. Available: http://world.std.com/~cme/html/web.html

[70] S. D. Kamvar, M. T. Schlosser, and H. Garcia-molina, "The eigentrust algorithm for reputation management in p2p networks," in *In Proceedings of the Twelfth International World Wide Web Conference*. ACM Press, 2003, pp. 640–651.

[71] O. Hallaraker and G. Vigna, "Detecting malicious javascript code in mozilla," in *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 85–94.

[72] The World Wide Web Consortium. (2009) Geolocation api specification. [Online]. Available: http://www.w3.org/TR/2009/WD-geolocation-API-20090707/

[73] S. Sire, M. Paquier, A. Vagner, and J. Bogaerts, "A messaging api for inter-widgets communication," in *WWW '09: Proceedings of the 18th international conference on World wide web*. New York, NY, USA: ACM, 2009, pp. 1115–1116.

[74] The World Wide Web Consortium. (2009) Widgets 1.0: Uri scheme. [Online]. Available: http://www.w3.org/TR/2009/WD-widgets-uri-20090618/

[75] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. (2008) Caja - safe active content in sanitized javascript. [Online]. Available: http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf

[76] Facebook. (2009) Fbjs. [Online]. Available: http://wiki.developers.facebook.com/index.php/FBJS

[77] M. L. Labs. (2008) Microsoft web sandbox. [Online]. Available: http://websandbox.livelabs.com/

[78] D. M. Scott Isaacs. (2008) Live labs web sandbox: Securing mash-ups, site extensibility, and gadgets. [Online]. Available: http://mschnlnine.vo.llnwd.net/d1/pdc08/PPTX/TL29.pptx

[79] D. Crockford. Adsafe. [Online]. Available: http://www.adsafe.org/

Appendix A

Markup and code examples

```
Python 2.5.1
>>> import hashlib
>>> import base64
>>> file = open("widget.wgt", "r")
>>> sha256 = hashlib.sha256()
>>> sha256.update(file.read())
>>> base64.b64encode(sha256.digest())
'...'
```

Figure A.1: Python reference implementation of the digest value calculation

```
Python 2.5.1 (requires M2Crypto)
>>> import M2Crypto
>>> import hashlib
>>> import base64
>>> file = open("widget.wgt", "r")
>>> sha256 = hashlib.sha256()
>>> sha256.update(file.read())
>>> privateKey = M2Crypto.RSA.load_key('privateKey.pem')
>>> signature = privateKey.sign(sha256.digest(), 'sha256')
>>> base64.b64encode(signature)
'...'
```

Figure A.2: Python reference implementation of the signature calculation using signature method RSA-SHA256

```
<widgets:resource
    xmlns="http://www.w3.org/2000/09/xmldsig#"
    xmlns:widgets="http://www.w3.org/ns/widgets">
  <Signature Id="DistributorASignature">
    <SignedInfo>
      <CanonicalizationMethod Algorithm="http://www.w3.org/2006/12/xml-c14n11"/>
      <SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
      <Reference URI="https://example.com/v1.1/widget.wgt">
        <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
        <DigestValue>...</DigestValue>
      </Reference>
    </SignedInfo>
    <Object Id="prop"><SignatureProperties xmlns:dsp="http://www.w3.org/2009/xmldsig-properties">
        <SignatureProperty Id="profile" Target="#DistributorASignature">
          <dsp:Profile URI="http://www.w3.org/ns/widgets-digsig#profile"/>
        </SignatureProperty>
        <SignatureProperty Id="role" Target="#DistributorASignature">
          <dsp:Role URI="http://www.w3.org/ns/widgets-digsig#role-distributor"/>
        </SignatureProperty>
        <SignatureProperty Id="identifier" Target="#DistributorASignature">
          <dsp:Identifier>07425f59c544b9cebff04ab367e8854b</dsp:Identifier>
        </SignatureProperty>
    </Object></SignatureProperties>
    <SignatureValue>...</SignatureValue>
    <KeyInfo><X509Data>
        <X509Certificate>...</X509Certificate>
    </KeyInfo></X509Data>
  </Signature>
</widgets:resource>
```

Figure A.3: An example of a resource file with one distributor signature