SYNGRESS®

XSS Attacks

CROSS SITE SCRIPTING EXPLOITS AND DEFENSE

XSS Is the New Buffer Overflow, JavaScript Malware Is the New Shell Code

- Learn to Identify, Exploit, and Protect Against XSS Attacks
- See Real XSS Attacks That Steal E-mails, Own Web Surfers, and Trojanize Backend Reporting Systems
- Leverage XSS Vulnerabilities to Allow Remote Proxy Attacks Into External and Internal Networks

Jeremiah Grossman
Robert "RSnake" Hansen
Petko "pdp" D. Petkov
Anton Rager
Seth Fogie **Technical Editor and Coauthor**

VISIT US AT

www.syngress.com

Syngress is committed to publishing high-quality books for IT Professionals and delivering those books in media and formats that fit the demands of our customers. We are also committed to extending the utility of the book you purchase via additional materials available from our Web site.

SOLUTIONS WEB SITE

To register your book, visit www.syngress.com/solutions. Once registered, you can access our solutions@syngress.com Web pages. There you may find an assortment of value-added features such as free e-books related to the topic of this book, URLs of related Web sites, FAQs from the book, corrections, and any updates from the author(s).

ULTIMATE CDs

Our Ultimate CD product line offers our readers budget-conscious compilations of some of our best-selling backlist titles in Adobe PDF form. These CDs are the perfect way to extend your reference library on key topics pertaining to your area of expertise, including Cisco Engineering, Microsoft Windows System Administration, CyberCrime Investigation, Open Source Security, and Firewall Configuration, to name a few.

DOWNLOADABLE E-BOOKS

For readers who can't wait for hard copy, we offer most of our titles in downloadable Adobe PDF form. These e-books are often available weeks before hard copies, and are priced affordably.

SYNGRESS OUTLET

Our outlet store at syngress.com features overstocked, out-of-print, or slightly hurt books at significant savings.

SITE LICENSING

Syngress has a well-established program for site licensing our e-books onto servers in corporations, educational institutions, and large organizations. Contact us at sales@syngress.com for more information.

CUSTOM PUBLISHING

Many organizations welcome the ability to combine parts of multiple Syngress books, as well as their own content, into a single volume for their own internal use. Contact us at sales@syngress.com for more information.

SYNGRESS®

This Page Intentionally Left Blank

XSS Attacks

CROSS SITE SCRIPTING EXPLOITS AND DEFENSE

Jeremiah Grossman
Robert "RSnake" Hansen
Petko "pdp" D. Petkov
Anton Rager

Seth Fogie Technical Editor and Co-Author

KEY	SERIAL NUMBER
001	HJIRTCV764
002	PO9873D5FG
003	829KM8NJH2
004	XVQ45LK89A
005	CVPLQ6WQ23
006	VBP965T5T5
007	HJJJ863WD3E
008	2987GVTWMK
009	629MP5SDJT
010	IMWQ295T6T

PUBLISHED BY
Syngress Publishing, Inc.
Elsevier, Inc.
30 Corporate Drive
Burlington, MA 01803

Cross Site Scripting Attacks: XSS Exploits and Defense

Printed in the United States of America
1 2 3 4 5 6 7 8 9 0

ISBN-10: 1-59749-154-3
ISBN-13: 978-1-59749-154-9

Publisher: Amorette Pedersen
Acquisitions Editor: Andrew Williams
Technical Editor: Seth Fogie
Page Layout and Art: Patricia Lupien
Copy Editor: Judy Eby
Cover Designer: Michael Kavish
Indexer: Richard Carlson

For information on rights, translations, and bulk sales, contact Matt Pedersen, Commercial Sales Director and Rights, at Syngress Publishing; email m.pedersen@elsevier.com.

Contributing Authors

Jeremiah Grossman founded WhiteHat Security in 2001 and is currently the Chief Technology Officer. Prior to WhiteHat, Jeremiah was an information security officer at Yahoo! responsible for performing security reviews on the company's hundreds of websites. As one of the world's busiest web properties, with over 17,000 web servers for customer access and 600 websites, the highest level of security was required. Before Yahoo!, Jeremiah worked for Amgen, Inc.

A 6-year security industry veteran, Jeremiah's research has been featured in USA Today, NBC, and ZDNet and touched all areas of web security. He is a world-renowned leader in web security and frequent speaker at the Blackhat Briefings, NASA, Air Force and Technology Conference, Washington Software Alliance, ISSA, ISACA and Defcon.

Jeremiah has developed the widely used assessment tool "WhiteHat Arsenal," as well as the acclaimed Web Server Fingerprinter tool and technology. He is a founder of the Website Security Consortium (WASC) and the Open Website Security Project (OWASP), as well as a contributing member of the Center for Internet Security Apache Benchmark Group.

For my family who puts up with the late nights, my friends who dare to test my PoC code, and everyone else who is now afraid to click.

Robert "RSnake" Hansen (CISSP) is the Chief Executive Officer of SecTheory. SecTheory is a web application and network security consulting firm. Robert has been working with web application security since the mid 90s, beginning his career in banner click fraud detection at ValueClick. Robert has worked for Cable & Wireless heading up managed security services, and eBay as a Sr. Global Product Manager of Trust and Safety, focusing on anti-phishing, anti-cross site scripting and anti-virus strategies. Robert also sits on the technical advisory board of ClickForensics and contributes to the security strategy of several startup companies. Before SecTheory, Robert's career fluctuated from Sr. Security Architect, to Director of Product Management for a publicly traded Real Estate company, giving him a great

breath of knowledge of the entire security landscape. Robert now focuses on upcoming threats, detection circumvention and next generation security theory.

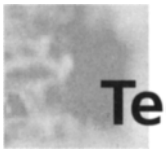
Robert is best known for founding the web application security lab at ha.ckers.org and is more popularly known as "RSnake." Robert is a member of WASC, IACSP, ISSA, and contributed to the OWASP 2.0 guide.

Petko "pdp" D. Petkov is a senior IT security consultant based in London, United Kingdom. His day-to-day work involves identifying vulnerabilities, building attack strategies and creating attack tools and penetration testing infrastructures. Petko is known in the underground circles as pdp or architect but his name is well known in the IT security industry for his strong technical background and creative thinking. He has been working for some of the world's top companies, providing consultancy on the latest security vulnerabilities and attack technologies.

His latest project, GNUCITIZEN (gnucitizen.org), is one of the leading web application security resources on-line where part of his work is disclosed for the benefit of the public. Petko defines himself as a cool hunter in the security circles.

He lives with his lovely girlfriend Ivana without whom his contribution to this book would not have been possible.

Anton Rager is an independent security researcher focused on vulnerability exploitation, VPN security and wireless security. He is best known for his WEPCrack tool, but has also authored other security tools including XSS-Proxy, WEPWedgie, and IKECrack. He has presented at Shmoocon, Defcon, Toorcon, and other conferences, and was a contributing technical editor to the book *Maximum Wireless Security.*



Technical Editor and Contributing Author

Seth Fogie is the Vice President of Dallas-based Airscanner Corporation where he oversees the research & development of security products for mobile platforms. Seth has co-authored several books, such as *Maximum Wireless Security, Aggressive Network Self Defense, Security Warrior,* and even contributed to *PSP Hacks.* Seth also writes articles for various online resources, including Pearson Education's InformIT.com where he is acting co-host for their security section. In addition, and as time permits, Seth provides training on wireless and web application security and speaks at IT and security related conferences and seminars, such as Blackhat, Defcon, and RSA.

This Page Intentionally Left Blank

Contents

Chapter 1

Cross-site Scripting Fundamentals

Solutions in this chapter:

- History of Cross-site Scripting
- Web Application Security
- XML and AJAX Introduction

- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions

Introduction

Cross-site scripting vulnerabilities date back to 1996 during the early days of the World Wide Web (Web). A time when e-commerce began to take off, the bubble days of Netscape, Yahoo, and the obnoxious blink tag. When thousands of Web pages were under construction, littered with the little yellow street signs, and the "cool" Web sites used Hypertext Markup Language (HTML) Frames. The JavaScript programming language hit the scene, an unknown harbinger of cross-site scripting, which changed the Web application security landscape forever. JavaScript enabled Web developers to create interactive Web page effects including image rollovers, floating menus, and the despised pop-up window. Unimpressive by today's Asynchronous JavaScript and XML (AJAX) application standards, but hackers soon discovered a new unexplored world of possibility.

Hackers found that when unsuspecting users visited their Web pages they could forcibly load any Web site (bank, auction, store, Web mail, and so on) into an HTML Frame within the same browser window. Then using JavaScript, they could cross the boundary between the two Web sites, and read from one frame into the other. They were able to pilfer usernames and passwords typed into HTML Forms, steal cookies, or compromise any confidential information on the screen. The media reported the problem as a Web browser vulnerability. Netscape Communications, the dominant browser vendor, fought back by implementing the "same-origin policy," a policy restricting JavaScript on one Web site from accessing data from another. Browser hackers took this as a challenge and began uncovering many clever ways to circumvent the restriction.

In December 1999, David Ross was working on security response for Internet Explorer at Microsoft. He was inspired by the work of Georgi Guninski who was at the time finding flaws in Internet Explorer's security model. David demonstrated that Web content could expose "Script Injection" effectively bypassing the same security guarantees bypassed by Georgi's Internet Explorer code flaws, but where the fault seemed to exist on the server side instead of the client side Internet Explorer code. David described this in a Microsoft-internal paper entitled "Script Injection." The paper described the issue, how it's exploited, how the attack can be persisted using cookies, how a cross-site scripting (XSS) virus might work, and Input/Output (I/O) filtering solutions.

Eventually this concept was shared with CERT. The goal of this was to inform the public so that the issue would be brought to light in a responsible way and sites would get fixed, not just at Microsoft, but also across the industry. In a discussion around mid-January, the cross organization team chose "Cross Site Scripting" from a rather humorous list of proposals:

- Unauthorized Site Scripting
- Unofficial Site Scripting
- Uniform Resource Locator (URL) Parameter Script Insertion

- Cross-site Scripting
- Synthesized Scripting
- Fraudulent Scripting

On January 25, 2000, Microsoft met with the Computer Emergency Response Team (CERT), various vendors (e.g., Apache, and so forth) and other interested parties at a hotel in Bellevue, WA to discuss the concept.

David re-wrote the internal paper with the help of Ivan Brugiolo, John Coates, and Michael Roe, so that it was suitable for public release. In coordination with CERT, Microsoft released this paper and other materials on February 2, 2000. Sometime during the past few years the paper was removed from Microsoft.com; however, nothing ever dies on the Internet. It can now be found at http://ha.ckers.org/cross-site-scripting.html

During the same time, hackers of another sort made a playground of HTML chat rooms, message boards, guest books, and Web mail providers; any place where they could submit text laced with HTML/JavaScript into a Web site for infecting Web users. This is where the attack name "HTML Injection" comes from. The hackers created a rudimentary form of JavaScript malicious software (malware) that they submitted into HTML forms to change screen names, spoof derogatory messages, steal cookies, adjust the Web page's colors, proclaim virus launch warnings, and other vaguely malicious digital mischief. Shortly thereafter another variant of the same attack surfaced. With some social engineering, it was found that by tricking a user to click on a specially crafted malicious link would yield the same results as HTML Injection. Web users would have no means of self-defense other than to switch off JavaScript.

Over the years what was originally considered to be cross-site scripting, became simply known as a Web browser vulnerability with no special name. What was HTML Injection and malicious linking are what's now referred to as variants of cross-site scripting, or "persistent" and "non-persistent" cross-site scripting, respectively. Unfortunately this is a big reason why so many people are confused by the muddled terminology. Making matters worse, the acronym "CSS" was regularly confused with another newly born browser technology already claiming the three-letter convention, Cascading Style Sheets. Finally in the early 2000's, a brilliant person suggested changing the cross-site scripting acronym to "XSS" to avoid confusion. And just like that, it stuck. XSS had its own identity. Dozens of freshly minted white papers and a sea of vulnerability advisories flooded the space describing its potentially devastating impact. Few would listen.

Prior to 2005, the vast majority of security experts and developers paid little attention to XSS. The focus transfixed on buffer overflows, botnets, viruses, worms, spyware, and others. Meanwhile a million new Web servers appear globally each month turning perimeter firewalls into swiss cheese and rendering Secure Sockets Layer (SSL) as quaint. Most believed JavaScript, the enabler of XSS, to be a toy programming language. "It can't root an operating system or exploit a database, so why should I care? How dangerous could clicking on a link

or visiting a Web page really be?" In October of 2005, we got the answer. Literally overnight the Samy Worm, the first major XSS worm, managed to shut down the popular social networking Web site MySpace. The payload being relatively benign, the Samy Worm was designed to spread from a single MySpace user profile page to another, finally infecting more than a million users in only 24 hours. Suddenly the security world was wide-awake and research into JavaScript malware exploded.

A few short months later in early 2006, JavaScript port scanners, intranet hacks, keystroke recorders, trojan horses, and browser history stealers arrived to make a lasting impression. Hundreds of XSS vulnerabilities were being disclosed in major Web sites and criminals began combining in phishing scams for an effective fraud cocktail. Unsurprising since according to WhiteHat Security more than 70 percent of Web sites are currently vulnerable. Mitre's Common Vulnerabilities and Exposures (CVE) project, a dictionary of publicly known vulnerabilities in commercial and open source software products, stated XSS had overtaken buffer overflows to become the number 1 most discovered vulnerability. XSS arguably stands as the most potentially devastating vulnerability facing information security and business online. Today, when audiences are asked if they've heard of XSS, the hands of nearly everyone will rise.

Web Application Security

The Web is the playground of 800 million netizens, home to 100 million Web sites, and transporter of billions of dollars everyday. International economies have become dependent on the Web as a global phenomenon. It's not been long since Web mail, message boards, chat rooms, auctions, shopping, news, banking, and other Web-based software have become part of digital life. Today, users hand over their names, addresses, social security numbers, credit card information, phone numbers, mother's maiden name, annual salary, date of birth, and sometimes even their favorite color or name of their kindergarten teacher to receive financial statements, tax records, or day trade stock. And did I mention that roughly 8 out of 10 Web sites have serious security issues putting this data at risk? Even the most secure systems are plagued by new security threats only recently identified as *Web Application Security*, the term used to describe the methods of securing web-based software.

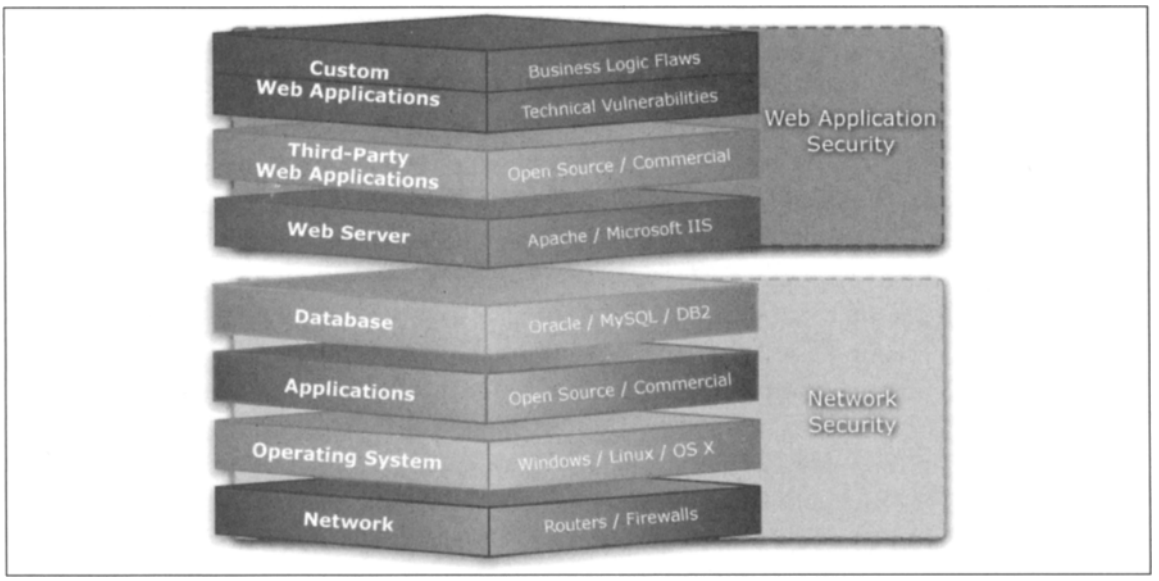
The organizations that collect personal and private information are responsible for protecting it from prying eyes. Nothing less than corporate reputation and personal identity is at stake. As vital as Web application security is and has been, we need to think bigger. We're beyond the relative annoyances of identity theft, script kiddy defacements, and full-disclosure antics. New Web sites are launched that control statewide power grids, operate hydroelectric dams, fill prescriptions, administer payroll for the majority of corporate America, run corporate networks, and manage other truly critical functions. Think of what a malicious compromise of one of these systems could mean. It's hard to imagine an area of information

security that's more important. Web applications have become the easiest, most direct, and arguably the most exploited route for system compromise.

Until recently everyone thought firewalls, SSL, intrusion detection systems, network scanners, and passwords were the answer to network security. Security professionals borrowed from basic military strategy where you set up a perimeter and defended it with everything you had. The idea was to allow the good guys in and keep the bad guys out. For the most part, the strategy was effective, that is until the Web and e-commerce forever changed the landscape. E-commerce requires firewalls to allow in Web (port 80 Hypertext Transfer Protocol [HTTP] and 443 Hypertext Transfer Protocol Secure sockets [HTTPS]) traffic. Essentially meaning you have to let in the whole world and make sure they play nice. Seemingly overnight the Internet moved from predominantly walled networks to a global e-commerce bazaar. The perimeter became porous and security administrators found themselves without any way to protect against insecure Web applications.

Web developers are now responsible for security as well as creating applications that fuel Web business. Fundamental software design concepts have had to change. Prior to this transformation, the average piece of software was utilized by a relatively small number of users. Developers now create software that runs on Internet-accessible Web servers to provide services for anyone, anywhere. The scope and magnitude of their software delivery has increased exponentially, and in so doing, the security issues have also compounded. Now hundreds of millions of users all over the globe have direct access to corporate servers, any number of which could be malicious adversaries. New terms such as cross-site scripting, Structured Query Language (SQL) injection, and a dozen of other new purely Web-based attacks have to be understood and dealt with.

Figure 1.1 Vulnerability Stack



Web application security is a large topic encompassing many disciplines, technologies, and design concepts. Normally, the areas we're interested in are the software layers from the Web server on up the vulnerability stack as illustrated in Figure 1.1. This includes application servers such as JBoss, IBM WebSphere, BEA WebLogic, and a thousand others. Then we progress in the commercial and open source Web applications like PHP Nuke, Microsoft Outlook Web Access, and SAP. And after all that, there are the internal custom Web applications that organizations develop for themselves. This is the lay of the land when it comes to Web application security.

One of the biggest threats that Web application developers have to understand and know how to mitigate is XSS attacks. While XSS is a relatively small part of the Web application security field, it possible represents the most dangerous, with respect to the typical Internet user. One simple bug on a Web application can result in a compromised browser through which an attacker can steal data, take over a user's browsing experience, and more.

Ironically, many people do not understand the dangers of XSS vulnerabilities and how they can be and are used regularly to attack victims. This book's main goal is to educate readers through a series of discussions, examples, and illustrations as to the real threat and significant impact that one XSS can have.

XML and AJAX Introduction

We are assuming that the average reader of this book is familiar with the fundamentals of JavaScript and HTML. Both of these technologies are based on standards and protocols that have been around for many years, and there is an unlimited amount of information about how they work and what you can do with them on the Internet. However, given the relatively new introduction of AJAX and eXtensible Markup Language (XML) into the Web world, we felt it was a good idea to provide a basic overview of these two technologies.

AJAX is a term that is often considered as being strongly related to XML, as the XML acronym is used as part of the name. That's not always the case. AJAX is a synonym that describes new approaches that have been creeping into Web development practices for some time. At its basics, AJAX is a set of techniques for creating interactive Web applications that improve the user experience, provide greater usability, and increase their speed.

The roots of AJAX were around long before the term was picked up by mainstream Web developers in 2005. The core technologies that are widely used today in regards to AJAX were initiated by Microsoft with the development of various remote-scripting techniques. The set of technologies that are defined by AJAX are a much better alternative than the traditional remote components such as the IFRAME and LAYER elements, defined in Dynamic Hyper Text Markup Language (DHTML) programming practices.

The most basic and essential component of AJAX is the *XMLHttpRequest* JavaScript object. This object provides the mechanism for pulling remote content from a server without the need to refresh the page the browser has currently loaded. This object comes in many

different flavors, depending on the browser that is in use. The *XMLHttpRequest* object is designed to be simple and intuitive. The following example demonstrates how requests are made and used:

```
// instantiate new XMLHttpRequest

var request = new XMLHttpRequest;

// handle request result

request.onreadystatechange = function () {
    if (request.readyState == 4) {

        //do something with the content

        alert(request.responseText);
    }
};

// open a request to /service.php

request.open('GET', '/service.php', false);

// send the request

request.send(null);
```

For various reasons, the *XMLHttpRequest* object is not implemented exactly the same way across all browsers. This is due to the fact that AJAX is a new technology, and although standards are quickly picking up, there are still situations where we need to resolve various browser incompatibilities problems. These problems are usually resolved with the help of AJAX libraries but we, as security researchers, often need to use the pure basics.

As we established previously in this section, the *XMLHttpRequest* object differs depending on the browser version. Microsoft Internet Explorer for example requires the use of *ActiveXObject('Msxml2.XMLHTTP')* or even *ActiveXObject('Microsoft.XMLHTTP')* to spawn similar objects to the standard *XMLHttpRequest* object. Other browsers may have different ways to do the exact same thing. In order to satisfy all browser differences, we like to use functions similar to the one defined here:

```
function getXHR () {
       var xhr = null;

       if (window.XMLHttpRequest) {
              xhr = new XMLHttpRequest();
       } else if (window.createRequest) {
              xhr = window.createRequest();
       } else if (window.ActiveXObject) {
              try {
                     xhr = new ActiveXObject('Msxml2.XMLHTTP');
              } catch (e) {
```

```
                    try {
                        xhr = new ActiveXObject('Microsoft.XMLHTTP');
                    } catch (e) {}
                }
        }

        return xhr;
};

// make new XMLHttpRequest object

var xhr = getXHR();
```

The XMLHttpRequest object has several methods and properties. Table 1.1 summarizes all of them.

Table 1.1 *XMLHttpRequest* Methods and Properties

Method/Property	Description
abort()	Abort the request.
getAllResponseHeaders()	Retrieve the response headers as a string.
getResponseHeader(name)	Retrieve the value of the header specified by name.
setRequestHeader(name, value)	Set the value of the header specified by name.
open(method, URL) *open(method, URL, asynchronous)* *open(method, URL, asynchronous, username)* *open(method, URL, asynchronous, username, password)*	Open the request object by setting the method that will be used and the URL that will be retrieved. Optionally, you can specify whether the request is synchronous or asynchronous, and what credentials need to be provided if the requested URL is protected.
onreadystatechange	This property can hold a reference to the event handler that will be called when the request goes through the various states.
readyState	The *readyState* parameter defines the state of the request. The possible values are: 0 – uninitialized 1 – open 2 – sent 3 – receiving 4 – loaded

Continued

Table 1.1 continued *XMLHttpRequest* Methods and Properties

Method/Property	Description
status	The status property returns the response status code, which could be 200 if the request is successful or 302, when a redirection is required. Other status codes are also possible.
statusText	This property returns the description that is associated with the status code.
responseText	The *responseText* property returns the body of the respond.
responseXML	The *responseXML* is similar to responseText but if the server response is served as XML, the browser will convert it into a nicely accessible memory structure which is also know as Document Object Model (DOM)

Notice the difference between the *responseText* and *responseXML* properties. Both of them return the response body, but they differentiate by function quite a bit.

In particular, *responseText* is used when we retrieve textual documents, HTML pages, binary, and everything else that is not XML. When we need to deal with XML, we use the *responseXML* property, which parses the response text into a DOM object.

We have already shown how the *responseText* works, so let's look at the use of *responseXML*. Before providing another example, we must explain the purpose of XML.

XML was designed to give semantics rather then structure as is the case with HTML. XML is a mini language on its own, which does not possess any boundaries. Other standards related to XML are XPath, Extensible Stylesheet Language Transformation (XSLT), XML Schema Definition (XSD), Xlink, XForms, Simple Object Access Protocol (SOAP), XMLRPC, and so on. We are not going to cover all of them, because the book will get quickly out of scope, but you can read about them at www.w3c.org.

Both XML and HTML, although different, are composed from the same building blocks that are known as elements or tags. XML and HTML elements are highly structured. They can be represented with a tree structure, which is often referred to as the DOM. In reality, DOM is a set of specifications defined by the World Wide Web Consortium, which define how XML structures are created and what method and properties they need to have. As we established earlier, HTML can also be parsed into a DOM tree.

One of the most common DOM functions is the *getElementsByTagName*, which returns an array of elements. Another popular function is *getElementById*, which return a single element based on its identifier. For example, with the help of JavaScript we can easily extract all <*p*> elements and replace them with the message "Hello World!." For example: