软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.006153]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

浏览器同源策略安全研究综述*

罗 武[1,3], 沈晴霓[2,3], 吴中海[2,3], 吴鹏飞[2,3], 董春涛[2,3], 夏玉堂[2,3]

[1](北京大学 信息科学技术学院,北京 100871)

[2](北京大学 软件与微电子学院,北京 100871)

[3](北京大学 软件工程国家工程中心,北京 100871)

通讯作者: 吴中海,沈晴霓, E-mail: wuzh@pku.edu.cn,qingnishen@ss.pku.edu.cn

摘 要: 随着云计算和移动计算的普及,浏览器应用呈现多样化和规模化的特点,浏览器的安全问题也日益突出.为了保证 Web 应用资源的安全性,浏览器同源策略被提出.目前,RFC6454、W3C 与 HTML5 标准都对同源策略进行了描述与定义,诸如 Chrome、Firefox、Safari 与 Edge 等主流浏览器均将其作为基本的访问控制策略.然而,浏览器同源策略在实际应用中,面临着无法处理第三方脚本引入的安全威胁、无法限制同源不同 frame 的权限、与其他浏览器机制协作时还会为不同源的 frame 赋予过多权限等问题,并且无法保证跨域/跨源通信机制的安全性以及内存攻击下的同源策略安全.该文对浏览器同源策略安全研究进行综述,介绍了同源策略的规则,并概括了同源策略的威胁模型与研究方向,主要包括同源策略规则不足及应对、跨域与跨源通信机制安全威胁及应对、以及内存攻击下的同源策略安全,并且展望了同源策略安全研究的未来发展方向.

关键词: 同源策略;浏览器安全;第三方脚本;跨源机制;内存攻击.

中图法分类号: TP309;TP393.092

中文引用格式: 罗武,沈晴霓,吴中海,吴鹏飞,董春涛,夏玉堂.浏览器同源策略安全研究综述.软件学报,2020,31. http://www.jos.org.cn/1000-9825/6153.htm

英文引用格式: Luo W, Shen QN, Wu ZH, Wu PF, Dong CT, Xia YT. State-of-the-Art survey of browser's same origin policy security. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/6153.htm

State-of-the-Art Survey of Browser's Same Origin Policy Security

LUO Wu[1,3], Shen Qing-Ni[2,3], WU Zhong-Hai[2,3], WU Peng-Fei[2,3], DONG Chun-Tao[2,3], XIA Yu-Tang[2,3]

[1](School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

[2](School of Software and Microelectronics, Peking University, Beijing 100871, China)

[3](National Engineering Research Center for Software Engineering, Peking University, Beijing 100871, China)

Abstract: With the popularity of cloud computing and mobile computing, browser applications show the characteristics of diversification and scale, and the browser security issues are increasingly prominent. To ensure the security of Web application resources, the browser's same-origin policy is proposed. Since then, the introduction of the same origin policy in RFC6454, W3C and HTML5 standards has driven modern browsers (e.g., Chrome, Firefox, Safari and Edge) to implement the same origin policy as the basic access control policy. The same-origin policy, however, in practice, faces the problems including handling security threats introduced by the third-party scripts, limiting the permissions of same-origin frames, assigning more permissions for cross-origin frames when they collaborate with browser's other mechanisms. It also cannot guarantee the safety of cross-domain or cross-origin communication mechanisms and the security under memory attacks. This paper reviews the existing researches on browser's same origin policy security. This paper firstly describes the same-origin policy rules, followed by summarizing the threat model for researches on same-origin policy and the research directions, including insufficient same-origin policy rules and defenses, attacks and defenses on cross-domain and

* 基金项目: 国家自然科学基金(61672062, 61232005)

收稿时间: 2020-04-26; 修改时间: 2020-08-13; 采用时间: 2020-09-15; jos 在线出版时间: 2020-10-12

cross-origin mechanisms, and same-origin policy security under memory attacks. Finally, this paper prospects the future research direction of browser's same-origin policy security.

Key words: same origin policy; browser security; third-party script; cross-origin schemes; memory attacks

浏览器的出现源自于互联网的发展.20 世纪 80 年代的互联网内容只有纯文本和纯文件,1989 年提出的万维网为互联网增加了内容布局、文本装饰与媒体等资源.丰富的互联网内容催生了浏览器的出现,早期的浏览器包括了 Cello、Mosaic 以及当时最受欢迎的 Netscape Navigator.随着网页动态性与面向用户的需求,Netscape Navigator 于 1994 年增加了"magic cookie"来区分用户,并且在 1995 年提出了改变网络世界的两个新特性:JavaScript 和文档对象模型(Document Object Model,简称 DOM).DOM 的提出使得网页 JavaScript 代码能够利用对应 API 来访问 HTML 文档的所有元素与属性、文档交互触发的事件以及 cookie 等资源.同时互联网内容的发展使得 HTML 文档能够更进一步地引入额外的资源,如其他网页文档或者媒体,这些资源又有自己对应的 cookie、DOM 与 JavaScript 等元素.JavaScript 与 HTML 文档的发展使得浏览器必须要提供一种方法来安全地实现代码与资源之间的交互.

浏览器的同源策略(Same Origin Policy,简称 SOP)被提出来定义浏览器加载的互联网资源的安全边界.Netscape 的工程师们在 1995 年发行的 Netscape Navigator 2 浏览器中实现了同源策略.2011 年发布的 RFC6454 标准明确了同源策略中"源" (origin)的定义[1],此后的 W3C 标准[2]与 HTML5 标准[3]也对同源策略进行了描述与定义.同源策略逐渐成为浏览器的基础访问控制策略,诸如 Chrome、Firefox、Safari 与 Edge 等主流浏览器都实现了同源策略[4,5].

同源策略将浏览器加载的所有资源都用一个称为"源"(origin)的字符串定义,该字符串由资源的协议(scheme)、服务器主机(host)和端口(port)组成,并且确保一个网页中的代码只能访问具有相同源的互联网资源.同源策略的策略实施发生在网页文档(称之为 frame)访问 Web 资源前,这些资源包括其他 frame 的 cookie、DOM 与 JavaScript、以及 Web 服务器的网络响应等.考虑用户访问了一个银行网站并在该网站上登录了自己账号,银行网站由于广告等其他需求可能包含了来自第三方的 frame.在没有同源策略的情况下,第三方 frame 的 JavaScript 代码将能够在用户不知情的情况下直接访问银行网站的资源,比如获取用户的银行卡信息与余额,甚至执行转账操作.而实施了同源策略的浏览器将会在第三方 frame 访问银行网站资源前进行同源检查,由于主客体属于不同源,浏览器将会拒绝该访问.同源策略的检查将能保证一个 Web 应用的资源(包含该 Web 应用上用户的隐私数据)不会被其他(不同源)Web 应用所访问,从而实现基于源的 Web 资源隔离.

同源策略的重要性使得其从诞生到现在一直都被认为是浏览器安全的基石.但是,同源策略在互联网演化过程中也暴露出来了诸多安全问题,包括同源策略规则不足所引起的安全威胁、跨域与跨源机制的安全威胁、以及内存攻击下的同源策略安全等.

同源策略的规则是浏览器安全中最受关注的部分之一,完备有效的同源策略规则是 Web 应用安全与用户隐私安全的基础.然而,随着 Web 的发展,Web 应用提出了更多的需求,这使得同源策略规则在提供这些便利性的同时损失了部分安全性.同源策略允许一个 frame 引入与执行第三方脚本,并且第三方脚本被视为与该 frame 相同的源而具备相同权限[6–8];同源策略赋予同源不同 frame 相同的权限,但由于不同的 URL 路径、在网页中的不同位置以及不同的历史行为,同源的不同 frame 也需要被赋予不同权限[9,10];同源策略还允许发送任意跨源网络请求,Web 服务器端不充分的网络请求授权检查将会导致资源泄漏[11,12].攻击者可以利用这些同源策略规则的不足来访问超出其权限之外的资源.

跨域与跨源通信机制是 Web 应用之间实现资源共享的主要方式,但也是攻击者用来绕过同源策略限制以恶意访问跨域或跨源资源的突破口之一.浏览器所提供的跨域与跨源通信 API 包含了 document.domain 与 postMessage.除此之外,Web 应用还可以利用 JSON-P 与跨域资源共享(Cross-Origin Resource Sharing,简称 CORS)机制来完成跨域与跨源通信.利用这些 API 与协议的脆弱性[13–16],攻击者将可以绕过同源策略的限制来访问跨源的资源.

内存攻击是破坏软件安全的重要攻击手段,浏览器作为软件也不可避免地面临着内存攻击.浏览器引入了

JavaScript 引擎(包括解释器与即时编译器)来执行网页的 JavaScript 代码.JavaScript 代码相当于 JavaScript 引擎的输入,恶意的输入有可能利用 JavaScript 引擎中的漏洞,从而实现恶意代码注入[17−19]、代码重用攻击[20,21]、以及仅数据攻击[22,23].成功发起基于内存的攻击意味着浏览器的功能受到破坏,攻击者将能绕过或者欺骗同源策略来直接访问跨源资源.

如何解决这些安全问题已经成为了学术界的研究热点.本文重点分析了现有浏览器同源策略安全的研究进展,深入讨论了同源策略规则的不足与应对方案;分析了跨域与跨源通信机制所引起的安全威胁与防御方案;并进一步讨论了内存攻击情况下的同源策略安全;最后,结合同源策略当前面临的安全问题,展望了该机制的未来发展方向.

1 同源策略概述

1.1 术语定义

网页资源的多样性使得浏览器环境中存在大量的术语,为便于描述,表 1 中总结了与同源策略相关的访问控制主客体术语定义.

Table 1 Descriptions for related terms

表 1 相关术语描述

术语		描述	HTML/JavaScript 代码示例
主体相关	frame	网页可以引入多个子网页来将自身划分为多个独立的部分,每个部分被称为 frame.被划分的网页可称为主 frame(main frame).相应的术语有:父 frame、子 frame、第三方 frame(与父 frame 不同源的子 frame)、frame 结构(网页中的所有 frame 及其关系)等	JavaScript 可通过 window.frames[i]来引用第 i 个子 frame,可通过 window.parent 来引用父 frame
	iframe	HTML 语言中的一个标签,一个 frame 可以通过该标签将另一个网页引入为其子 frame	HTML: <iframe src="URL of the child frame" />
	window 对象	每个 frame 具有对应的 window 对象,可以视为该 frame 的 JavaScript 代码的 this 指针	JavaScript 可通过 window.foo 来获取该 frame 的全局变量 foo
	第三方脚本	每个 frame 可以通过 HTML 语言的 script 标签来引入并执行其他源的 JavaScript 脚本,这些脚本被称为第三方脚本	HTML: <script src="URL of the third-party script" />
	宿主 frame (host frame)	承载第三方脚本的 frame 被称为宿主 frame	N/A
客体相关	DOM 对象	在渲染网页时, 浏览器将 HTML 元素转变为文档对象模型(Document Object Model,简称 DOM)存储	JavaScript 可通过 document.getElementById("ID")等 API 来获取 DOM 对象的引用
	JavaScript 对象	Frame 中的 JavaScript 代码执行过程中所产生的对象	N/A
	网络资源	Frame 中的 JavaScript 代码能够发起网络请求来与服务器进行通信	JavaScript 可通过 XMLHttpRequest API 来发送网络请求与读取响应
	cookie	由 Web 应用服务器与浏览器共同维护的资源,通常在用户登录 Web 应用后设置,可用来进行用户身份认证	JavaScript 可通过 document.cookie API 访问 cookie
	本地存储 (local storage)	浏览器为 Web 应用所维护的本地资源,以键值对形式存在	JavaScript 可通过 localStorage API 访问本地存储
	会话存储 (session storage)	与本地存储相似,但为临时存储,在浏览器关闭后将会被删除	JavaScript 可通过 sessionStorage API 访问会话存储

1.2 同源策略规则

同源策略(Same Origin Policy,简称 SOP)是浏览器安全的基础安全策略[5].浏览器为每个加载的 frame 设置一个称为"源"(origin)的属性.区别于网站(site)与域名(domain),源是一个三元组:<scheme, host, port>.其中

scheme 是指网址的协议,如 http 或 https 等;host 是指服务器主机在因特网上的域名,与 domain 不同,host 涵盖了子域名,如 abc.xyz.com;port 是指服务器提供网站服务的端口,如 8080.

同源策略的检测要求是“同源”,即主体与客体所在 frame 的源的三元组需要完全一致.举例来说,frame https://abc.xyz.com:8080/index.html 与 http://abc.xyz.com:8080/index.html、https://xyz.com:8080/index.html、https://abc.xyz.com:80/index.html 等 frame 分别在协议、域名与端口上不一致,故而不能访问这些 frame 的资源.

同源策略的目标在于实现基于源的 Web 资源隔离,以避免一个恶意 Web 应用获取或修改其他不同源应用的资源或其上的用户隐私数据.图 1 描述了同源策略的基本规则.同源策略的规则与客体资源类型有关,具体来说(如图 1 中数字标注所示):

(1) DOM 对象、JavaScript 对象、cookie、本地存储与会话存储:JavaScript 代码不能访问不同源网页的该类资源,如图 1 中来自 alice.com 与 bob.com 的两个 frame 无法访问对方的该类资源;

(2) 网络请求与网络响应:同源策略不限制 JavaScript 代码发送网络请求,但是只有来自同源的服务器返回的网络响应才能被读取.如图 1 所示,来自 https://alice.com 的 frame 中的 JavaScript 代码能够向 https://bob.com 的服务器发送数据,但是该服务器返回的数据不能被 alice.com 网页读取;

(3) 第三方资源:一个网页可以嵌入第三方资源,包括 JavaScript 脚本、层叠样式表(Cascading Style Sheets,简称 CSS)与图片等.同源策略视网页本身的源为这些资源的源.在图 1 中,来自 https://alice.com 的 frame 通过<script src=“https://bob.com/script.js”>标签来引入来自 bob.com 的第三方脚本,该脚本能够以 alice.com 的身份来执行,但其脚本内容对于 alice.com 里的 JavaScript 代码来说是不可读的.

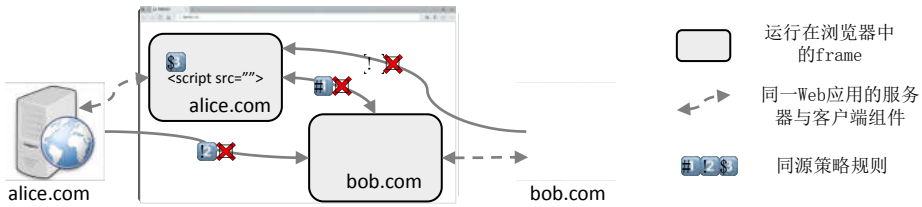


Fig 1 Rules of Same Origin Policy

图 1 同源策略规则

2 同源策略安全研究方向

2.1 威胁模型

2.1.1 研究场景

随着 Web 的发展,浏览器已经逐渐演化为一个多主体的操作环境.通常来说,每个网站包含服务器程序与在用户浏览器中运行的 frame.Frame 可以认为是含有代码和数据的主体.代码包括 frame 的 HTML 与 JavaScript 代码,而数据资源包括运行时的 JavaScript 对象、DOM 对象、cookie 及本地存储等.作为网页代码的执行环境,浏览器实施了同源策略来控制 frame 代码对数据资源的访问.同源策略研究的威胁模型包含了以下三类角色:

- 受害者网站:受害者网站诚实地按照指定的隐私与安全协议执行其业务逻辑,用户在该网站具备有对应的账号,从而具备相应的隐私数据(如银行网站中的账单与余额等)及私有操作(如银行网站中的转账操作);
- 攻击者:攻击者在因特网上拥有一个合法的域名,攻击者在该域名上布置了相应的恶意网页来访问受害者网站(包括服务器与客户端网页)的资源、或者向受害者网站的 frame 提供恶意的第三方 JavaScript 脚本.攻击者网站应当与受害者网站是不同源的,因此理应受到同源策略的保护;
- (受害者)用户:用户被诱导在某一时刻使用其浏览器访问了攻击者网页或者含有攻击者脚本的受害者网页,使得攻击者的代码在用户浏览器中运行.

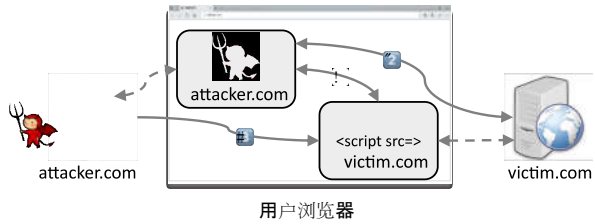


Fig 2 Threat model for researches on same-origin policy, including three roles and three attack scenarios

图 2 同源策略研究的威胁模型,包括三类角色与三种攻击场景

如图 2 所示,攻击者的恶意代码获取用户在受害者网站上资源的研究场景有三种:

(1) 用户在其浏览器上访问了攻击者网站网页(如图 2 中 attacker.com)与受害者网站网页(如图 2 中 victim.com),当攻击者和受害者网页位于同一个浏览实例(browsing instance)[24,25]时,攻击者代码将能够获得受害者网站网页的引用;若同源策略被破坏,攻击者代码将能通过该引用来访问受害者网站网页的资源,包括 DOM、JavaScript 与 cookie 等,若用户已经在受害者网站网页中登录,攻击者代码将能获取用户在受害者网站中的隐私数据与执行私有操作;

(2) 用户在其浏览器上访问了攻击者网站网页,攻击者意图伪装成用户向受害者网站服务器发送恶意的网络请求;该场景要求用户事先在受害者网页中进行了登录,并且在攻击发生时用户在受害者网站中的 cookie 没有过期;

(3) 用户在其浏览器上访问了受害者网站网页,受害者网站网页错误地引用了攻击者网站的资源,如引用了攻击者网站的第三方 JavaScript 脚本来完成诸如广告、网站分析等功能.

特别的,第一类场景需要攻击者网页与受害者网页处于同一浏览实例中,存在两种途径来实现该目标:

- 攻击者网页内嵌(或弹出)受害者网页:攻击者通过钓鱼攻击等方式,使得受害者用户访问了攻击者域名下的某个网址,该恶意网址对应的攻击者网页利用 HTML 中的 iframe 标签或 window.open API 嵌入了受害者网页作为子 frame.注意攻击者可以将子 frame 设置为不可见,使得受害者用户无法直观观察到;
- 受害者网页内嵌(或弹出)攻击者网页:用户正常地访问了受害者网站的网址,该受害者网页为完成诸如用户追踪与分析、网页性能分析等任务,主动嵌入了攻击者网页作为第三方 frame.

2.1.2 敌手模型

根据以上研究场景,当攻击者的恶意代码在受害者用户的浏览器中执行时,攻击者将尝试绕过同源策略来访问用户在不同源的受害者网站中的资源.根据攻击者所具备的攻击能力,我们定义两种不同的敌手模型:

(1) 策略攻击者:攻击者了解浏览器同源策略的规则,尝试找到同源策略规则中未覆盖到的行为或者利用现有跨源/跨域通信机制来绕过同源策略,以成功访问受害者网页的资源.若用户在受害者网页中进行了登录,或者在攻击发生时用户的 cookie 没有过期,策略攻击者将能够访问用户在受害者网页中的隐私数据,甚至伪装为用户向受害者网站服务器发送恶意请求(如转账);

(2) 内存攻击者:攻击者了解浏览器中的 JavaScript 引擎的原理,并且该 JavaScript 引擎中存在漏洞使得攻击者能够具备有操作内存的权限,如修改浏览器的内存数据,甚至于发起代码注入或者代码重用攻击来以浏览器的身份执行任意代码.内存攻击者可以认为能够攻击浏览器,在这种情况下,浏览器的同源策略很容易被绕过.

注意本文仅考虑针对浏览器同源策略安全的相关攻防机制,因此并不考虑由于受害者网站网页代码的漏洞(如跨站脚本攻击 XSS[26−29])所导致的跨源资源访问与数据泄漏.特别的,由于操作系统所提供了进程间的内存隔离,本文敌手模型中的内存攻击者必须要求攻击者代码与受害者网页位于同一个进程内.攻击者通过发起内存攻击来控制该进程,进而绕过或欺骗同源策略来访问受害者网页资源.根据浏览器的多进程模型[24,25],网页代码在渲染进程(renderer process)中执行,并且利用操作系统提供的沙箱机制对渲染进程进行限制,包括提供独

立的命名空间甚至限制允许调用的系统调用.因此渲染进程将不得不通过 IPC 消息与位于后台的浏览器内核进程(browser kernel process)通信来完成特权操作,如操作文件系统、网络以及传感器等.然而,浏览器的多进程模型仍然把属于同一个浏览实例的 frame 放置在同一个渲染进程中.据此,内存攻击者通常存在于图 3 所示的第一类研究场景中.

2.2 研究方向

浏览器中不同主体之间数据的隐私保护与代码的隔离执行归功于同源策略.同源策略实施的有效性极大地影响浏览器的安全,因而大量的研究者们对同源策略进行了研究.我们研究与分析了近年来与 Web 相关领域的顶级会议(S&P、CCS、Usenix Security、NDSS、OSDI 以及 WWW 等)上与同源策略有关的论文.这些研究现状可以分为以下三部分:

(1) 同源策略规则不足及应对(见第 3 节):在策略攻击者存在的情况下,考虑同源策略规则有可能存在的不足与缺陷,分析这些不足可能导致的后果,并给出相应的弥补方案.例如,同源策略无法处理第三方脚本的安全威胁、无法限制同源不同 frame 的权限、与其他浏览器机制协作时还会为不同源的 frame 赋予过多权限等;

(2) 跨域/跨源通信支持安全威胁及应对(见第 4 节):这一类研究方向考虑浏览器允许的跨域/跨源通信机制,针对这些合法的通信方式进行安全分析,发现其中存在的安全威胁并提供相应的防御方案;

(3) 内存攻击下的同源策略安全(见第 5 节):这一类研究方向考虑在浏览器中发起内存攻击的可能性,并基于此给出相应的内存攻击方案与防御措施.在内存攻击者存在的情况下,同源策略将无法实施有效的资源访问控制,需要借助一些内存防御方案来进行抵御.

我们分析总结了这三类同源策略安全研究方向的攻击者类型、攻击原理、防御机制以及代表性研究工作,表 2 给出了总结分析结果.

Table 2 Three research directions of same origin policy security

表 2 同源策略安全的三类研究方向

研究方向		攻击者类型	攻击原理概述	防御机制概述	代表性研究工作
同源策略规则不足及应对	第三方脚本的过度授权与防御	策略攻击者	同源策略允许 frame 嵌入第三方脚本并赋予其与 frame 相同的权限	隔离第三方脚本并提供安全的资源共享机制	Nikiforakis 等[6]; Ingram 等[30]
	同源不同 frame 的过度授权与防御		同源策略赋予同源的不同 frame 相同权限,然而存在不同 frame 不同权限的需求	设计与实现细粒度或动态的同源策略,并增强其他相关浏览器机制	Cao 等[9]; Lerner 等[31]
	不同源 frame 的过度授权与防御		同源策略允许嵌入第三方脚本与发送任意网络请求,与 cookie 机制的配合将能破坏同源策略	为网络请求增加灵活的策略规则或增强 cookie 机制	Barth 等[11]; Lekies 等[12]
跨域/跨源通信机制安全威胁及应对	无服务器辅助的跨域/跨源通信机制与防御	策略攻击者	跨域/跨源 API 所引起的机密性破坏与同源策略规则的不一致性	提供应用程序级别的 API 安全封装或安全使用检测	Singh 等[13]; Son 等[14]
	服务器辅助的跨域/跨源通信机制与防御		跨域/跨源协议所引起的机密性与完整性破坏	修复跨域/跨源协议漏洞与进行安全增强	Popescu[15]; Chen 等[16]
内存攻击下的同源策略安全	代码注入攻击与防御	内存攻击者	注入任意代码来绕过同源策略访问跨源资源	W^X、地址空间随机化	Song 等[17]; Blazakis 等[19]
	代码重入攻击与防御		利用代码重用攻击以获得任意代码执行能力来绕过同源策略限制	常数盲化、地址空间随机化、控制流完整性	Snow 等[32]; Athanasakis 等[20]
	仅数据攻击与防御		修改安全相关元数据来欺骗或绕过同源策略限制	内存隔离、地址空间随机化	Jia 等[22]; Reis 等[33]

3 同源策略规则不足及应对

浏览器同源策略的目的是实现基于源的资源隔离,这些资源涵盖了 JavaScript 与 DOM 对象、cookie、本地存储以及服务器返回的网络响应等.同源策略规则的不足,将使得攻击者能够绕过同源策略来非法获取不同源的资源.同源策略规则的不足体现在以下三个方面:

(1) 同一 frame 中第三方脚本的过度授权:一个 frame 中可能包含了第三方脚本.同源策略用承载它们的 frame(称之为宿主 frame)的源来判断这些第三方脚本可以访问的资源,这使得恶意的第三方脚本能够自由地访问宿主 frame 所在源的任意资源;

(2) 同源不同 frame 的过度授权:同源策略为处于同一个源下的所有 frame 赋予相同的权限,然而,有些 Web 应用的特性使得同源不同 frame 需要进行必须的隔离.例如在博客网站中,xyz.com/a/与 xyz.com/b/分别代表了两个用户 a 和 b 的主页,对它们进行资源隔离是有必要的,而同源策略并不能适应这些场景;

(3) 不同源 frame 的过度授权:目前浏览器在发送网络请求时,会默认将用户的 cookie 携带到网络请求中,一个恶意 frame 可以通过利用 cookie 机制来修改不同源服务器的状态,甚至是获取不同源的资源.

本节将对以上三类问题的现有研究工作进行总结分析,包括相应的攻击场景、安全威胁与防御方案.

3.1 第三方脚本的过度授权与防御

当 Web 应用程序的开发人员决定引入来自第三方提供者的 JavaScript 脚本时,同源策略不允许第三方脚本被宿主 frame 读取,但是允许它以宿主 frame 的权限来执行(见 1.2 节).一方面来说,恶意的第三方脚本提供者可以直接访问宿主 frame 的所有资源,包括偷取敏感数据或修改网页内容;另一方面来说,非恶意的但是存在漏洞的第三方脚本为攻击者攻击宿主 frame 提供了便利:攻击者可以以第三方脚本为跳板来实现对宿主 frame 的攻击.本小节将首先介绍宿主 frame 引入第三方脚本的目的与第三方脚本的分类,然后分析第三方脚本过度授权所带来的安全威胁,最后总结讨论现有防御方案的优缺点.

3.1.1 用途与分类

为了丰富功能与增强交互性,Web 应用广泛集成第三方脚本到其自身的网页中. Yue 等[34]发现 96.9%的网站包含来自其他源的脚本,平均每个网站包含 3.1 个第三方脚本.Nikiforakis 等[6]更进一步地统计了 Alexa top 10k 网站中 10 个被引入最多的第三方脚本,并记录了这些第三方脚本提供的服务,以及在前 10k 个 Alexa 网站中的使用率.如表 3 所示,主流的第三方脚本通常提供了诸如 Web 分析(如 Google Analytics 和 addthis)、动态广告(如 Google AdSense)、社交网络(如 Facebook 和 Twitter)与用户追踪(如 Quantserve)等服务.有些第三方脚本还提供了编程库(如 jQuery)与密码算法库(如 CryptoJS)等服务.

Table 3 The ten most popular third-party scripts used by Alexa top 10,000 Internet web sites[6]

表 3 Alexa 10k 网站中 10 大最受欢迎的第三方脚本[6]

第三方脚本	提供的服务	在 Alexa top 10k 网站中的占比
www.google-analytics.com/ga.js	Web 分析	68.37%
pagead2.googlesyndication.com/pagead/show_ads.js	动态广告	23.87%
www.google-analytics.com/urchin.js	Web 分析	17.32%
connect.facebook.net/en_us/all.js	社交网络	16.82%
platform.twitter.com/widgets.js	社交网络	13.87%
s7.addthis.com/js/250/addthis_widget.js	社交网络与 Web 分析	12.68%
edge.quantserve.com/quant.js	Web 分析与用户追踪	11.98%
b.scorecardresearch.com/beacon.js	市场研究	10.45%
www.google.com/jsapi	Google 辅助功能	10.14%
ssl.google-analytics.com/ga.js	Web 分析	10.12%

3.1.2 安全威胁

一个 frame 引入大量的第三方脚本意味着该 frame 具有更大的攻击面.我们首先来总结分析由引入第三方脚本所带来的对宿主 frame 的新的攻击向量,包括以下三种.

第一类是第三方脚本劫持,即网页开发者在编写宿主 frame 时可能出现编写错误,攻击者可以利用这些编写错误来使宿主 frame 加载与运行攻击者的第三方脚本,进而攻击宿主 frame.Nikiforakis 等[6]于 2012 年对 Alexa top 10k 的网站进行了分析,发现 88.45%的网站至少包含一个第三方脚本,最坏情况下有网站从 295 个第三方服务器中加载了脚本,并且有 0.27%的第三方脚本是通过 IP 地址引入的.这意味着攻击者可以采取以下方式来使得宿主 frame 运行其恶意代码:

- 若宿主 frame 通过使用 localhost 来引入第三方脚本,并且用户机器为多用户时,恶意用户可以创建搭载在本地的 Web 服务器来攻击;
- 若宿主 frame 通过使用私有 IP 地址来引入第三方脚本,位于内网内的攻击者可以创建对应的 Web 服务器来发起攻击;
- 若宿主 frame 引入第三方脚本的域名或 IP 地址已经过期,攻击者可以注册该过期域名/IP 来发起攻击;
- 若开发者在编写第三方脚本域名时出现编写错误,如将 googlesyndication.com 错误书写为 googlesyndicatio.com,攻击者可以注册该错误的域名来发起攻击.

第二类是利用第三方脚本的漏洞,即攻击者可以直接攻击第三方脚本来达到攻击宿主 frame 的目标.Stock 等[35]于 2015 年对 FireFox 浏览器进行修改以增加网页的数据流检测.攻击者可以利用跨站脚本攻击来将恶意的数据转变为 JavaScript 代码来在宿主 frame 中执行.根据其实验数据,在 Alexa top 10k 网站中检测到 1273 个导致跨站脚本攻击的漏洞,而其中有 273 个漏洞来自于第三方脚本、165 个漏洞来自于第三方脚本与宿主 frame 自身代码的组合.Lauinger 等[36]于 2017 年对 Alexa top 75k 网站中所包含的 11,141,726 个第三方脚本进行了分析,通过爬取脆弱性数据库、Github 评论与 release note 等数据来判断第三方脚本是否存在脆弱性.实验结果发现 37%的网站至少包含一个已知漏洞版本的第三方脚本.此外,有些网站还使用了很早版本的第三方脚本,这些过时版本与最新版本的平均滞后时间为 1177 天.新的版本可能包含了对之前版本漏洞的修复或者增加一些机制来增强脚本的安全性,使用过时的第三方脚本为攻击者提供了额外的攻击向量来攻击宿主 frame.同时,由于浏览器同源策略赋予了第三方脚本与宿主 frame 相同的访问权限,攻击者可以攻击这些脆弱的第三方脚本来访问宿主 frame 所在源中的资源.

第三类是利用第三方脚本的间接引用特性,即第三方脚本可以继续引入其他脚本,而宿主 frame 开发者通常并没有意识到这些外部脚本的存在.Kumar 等[37]于 2017 年对 Alexa top 1,000k 的网站进行了分析,他们将宿主 frame 明确引入的第三方脚本称之为显式依赖(explicit trust),而由这些第三方脚本引入的除该第三方之外的源的脚本称为隐式依赖(implicit trust).实验结果表明 9%的网站加载了隐式依赖的外部脚本,而 top 100k 的网站中有 13%运行了隐式依赖的外部脚本.Ikram 等[8]除了分析 Alexa top 200k 网站中的隐式依赖脚本外,还更近一步地对这些隐式的第三方提供商进行分类.借助于现有的 VirusTotal 工具并设定合适的阈值,Ikram 等发现 1.2%的隐式第三方是可疑的,且 73%的网站从这些可疑的隐式第三方中加载了资源.这些现有的实验数据意味着攻击者可以不用直接去攻击一个 frame 或者其显式依赖,而是通过攻击一些隐式的第三方脚本提供商来完成攻击.

上述三类威胁场景对宿主 frame 所造成的攻击后果包括实现用户追踪、数据泄漏与篡改、阻止 HTTPS 部署、以及破坏宿主 frame 业务逻辑等.Zhou 等[7]开发了 ScriptInspector,一个修改后的浏览器用来拦截、记录与检查第三方脚本对关键资源的访问.通过对 Alexa top 200 US 网站的实验,发现几乎所有的第三方脚本访问了浏览器的基本属性,包括 navigator、screen 以及 DOM 中的根节点(document 与 body),第三方脚本可以利用这些属性来实现浏览器指纹功能[38–42],进而完成用户追踪而破坏用户隐私;Zhou 等还发现大部分的第三方脚本都发送了网络请求,并且有些网络请求还发送给了其他源的服务器,这可能会造成数据泄漏;一些广告和社交第三方脚本还存在对宿主 frame 的内容进行了修改与读取的情况,如 Zhou 等的实验中发现有脚本读取了用户购物车信

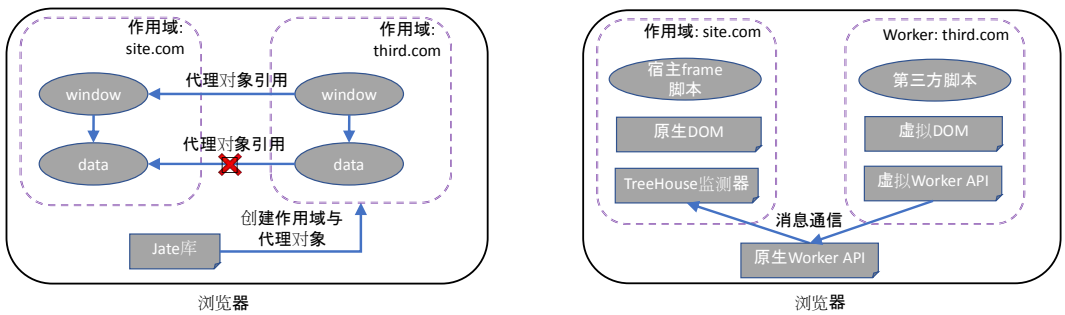
息;Kumar 等[37]的实验发现在前 100 万个只使用 HTTP 协议的网站中,有 55%的 HTTP 网站依赖于不支持 HTTPS 访问的内容,由于浏览器不会执行通过 HTTP 协议加载到 HTTPS frame 中的内容,这些网站因而不能迁移为 HTTPS 部署;Patra 等[43]还发现宿主 frame 自身的脚本与第三方脚本有可能存在冲突,由于宿主 frame 中的所有脚本具有同样的 JavaScript 作用域,第三方脚本可以声明在宿主 frame 中已经存在的全局变量.当宿主 frame 之后调用该变量时,JavaScript 引擎将会使用最新加载的脚本所定义的变量,之前的变量将会被覆盖,这有可能直接影响宿主 frame 自身的业务逻辑.

3.1.3 防御方案

应对第三方脚本安全威胁的主要方案是,将宿主 frame 本身的脚本与第三方脚本运行在不同的作用域下,以实现第三方脚本与宿主 frame 的隔离.根据隔离机制方案的不同,我们将其划分为以下四类.

第一类是基于语言的防御机制,即利用 JavaScript 语言的特性来实现脚本之间的隔离与资源共享.在该方向上的防御机制包括 Jate[44]、JSand[45]、ScriptProtect [46]、BrowserShield[47]、Caja[48]与 GateKeeper[49]等.图 3(a)展示了 Tran 等设计的 Jate 系统[44].该系统首先利用一个网络代理来将在网页中包含一个 Jate 库并对第三方脚本进行重写.Jate 库利用 JavaScript 的“with(scope){}”语句使得第三方脚本的所有变量不能超过 scope 变量的作用域,因而第三方脚本无法直接访问宿主 frame 的其他变量.为支持第三方脚本对宿主 frame 的正常操作,Jate 利用 JavaScript 的对象代理(proxy)机制为宿主 frame 中的相关变量创建代理对象,并在第三方脚本重写时允许对这些代理对象的访问.同时由于对象代理机制允许设置回调函数,宿主 frame 可以实施对应的策略:当一个代理对象被调用时,该对象上的策略回调函数将会被调用,用来判断第三方脚本是否被授权来访问宿主 frame 的对象,如图 3(a)中允许对 window 对象的引用,但拒绝对 data 对象的引用.其实验表明 Jate 机制引入了接近 20%的开销.

第二类是基于 Worker 的防御机制,即利用浏览器提供的 Web Worker 来实现宿主 frame 与第三方脚本的隔离.由于 JavaScript 的单线程性质,frame 中的比较耗时的操作将会很大影响网页浏览体验.Web Worker 机制使得 JavaScript 可以请求浏览器创建新的线程来运行指定代码.浏览器为 Web Worker 中的代码与宿主 frame 其他代码创建不同的作用域而实现隔离. Ingram 等据此设计了 TreeHouse 系统[30].如图 3(b)所示,该系统将第三方脚本放置在 Web Worker 中,并创建虚拟 DOM 来允许第三方脚本对特定 DOM 的合法访问.第三方脚本与宿主 frame 之间的交互需要通过消息通信机制来实现.



(a) 第一类防御机制:Jate 系统[44] (b) 第二类防御机制:TreeHouse 系统[30]

Fig 3 Example for the first two types of defense mechanisms

图 3 第一类与第二类防御机制示例

第三类是基于 frame 的防御机制,即利用浏览器的同源策略来实现宿主 frame 与第三方脚本的强隔离.在该方向上的防御机制包括 AdJail[50]、Pivot[51]、Mashic[52]与 MashupOS[53]等.举例来说,Louw 等提出 AdJail[50]系统来将第三方脚本放置在一个具有唯一源的 frame 中.根据同源策略,运行在另一个源中的第三方脚本将无法直接访问原来宿主 frame 的数据.为了保留第三方脚本对宿主 frame 某些资源的访问,AdJail 系统利用消息通信机

制来实现资源共享.Mickens 等更进一步地提出了 Pivot 系统[51].类似于 AdJail,第三方脚本被放置在不同源的 frame 中来实现隔离.然而,AdJail 的资源共享由于利用消息通信机制使得其必须是异步操作,Pivot 利用 JavaScript 语言中的 generator 特性来对消息通信进行封装,实现同步的资源共享.

第四类是基于 JavaScript 引擎的防御机制,即通过对 JavaScript 引擎进行修改以实现宿主 frame 与第三方脚本的隔离.Dong 等提出了基于 JavaScript 引擎的防御机制 AdSentry 系统[54].AdSentry 在浏览器中运行了一个影子 JavaScript 引擎.第三方脚本将运行在该影子 JavaScript 引擎中,从而实现与宿主 frame 的隔离.第三方脚本与宿主 frame 的资源共享也需要利用消息通信机制来实现.Meyerovich 等与 Acker 等先后提出了 ConScript[55]与 WebJail[56]来限制第三方资源.具体来说,通过在 JavaScript 引擎中增加 API 来对原来 JavaScript 函数引入新的建议函数(advice function).当建议函数存在时,JavaScript 引擎将会调用建议函数,否则调用原函数.宿主 frame 因此可以利用建议函数来限制第三方脚本.

以上四类方案的基本思路归为两个角度:

- 实现语言上的隔离.第一类防御机制利用 JavaScript 语言的特性或者提出新的语言来实现隔离.这种方式的优势在于便于资源共享,但是在可用性上是存在不足的.具体来说,这些机制需要对 JavaScript 所有可能的对象引用都进行限制,JavaScript 语言的复杂性与动态性使得策略仲裁很难确保完备性.而且 JavaScript 与 HTML 在不断地更新,甚至不同浏览器在 JavaScript API 实现上还可能有区别,这必然导致这些机制很难实际采用.此外对第三方脚本的重写还涉及到动态代码,如 eval 等,这将造成很大的运行时开销;
- 实现系统上的隔离.后三类防御机制能实现系统上的隔离,其优势在于 JavaScript 与 HTML 的复杂性与更新不会影响隔离机制.这些机制可以粗略地认为它们将第三方脚本转变为了第三方 frame,这将导致直接函数调用与对象引用等能力的缺失.为保证第三方脚本的正常操作,这些机制必须提供额外的消息通信机制(如 postMessage[57])来允许第三方脚本与宿主 frame 之间的资源共享.但使用消息通信机制会带来一些不足,比如第三方脚本原本的同步操作经转变为了异步操作,甚至可能需要第三方脚本自己阻塞来等待响应消息的到来(如 Pivot[51]).这种异步操作不仅会带来很大的性能开销,还需要考虑是否对网页的业务逻辑造成影响.

3.2 同源不同frame的过度授权与防御

浏览器同源策略的策略实施依据为主体与客体的源是否一致,这意味着同源的不同 frame 将拥有相同的访问控制权限,即都能访问该源的任意资源.然而,同源的不同 frame 在一些情况下需要赋予不同的权限.在这些情况下,由于权限受限的 frame 可能通过访问同源且授权的 frame 来提升权限,同源策略将使得这些 frame 的权限控制形同虚设.本小节将首先分析同源不同 frame 需要赋予不同权限的场景与对应的安全威胁,然后总结讨论现有的一些防御方案的优缺点.

3.2.1 安全威胁

在一个网页中,同源的两个不同的 frame 并不一定是完全对等的,因而需要被赋予不同权限,主要包括以下三类情况.

第一类是同源但不同 URL 路径的 frame.Moshchuk 等[58]指出同源策略无法实现在 URL 路径上的隔离,考虑分别来自于 https://www.facebook.com/user1 与 https://www.facebook.com/user2 的两个同源 frame,它们代表着不同用户的主页,但却无法实现权限上的隔离.Cao 等[9]认为同源策略需要支持更进一步的主体隔离.例如,一个 Web 应用包含有一个良好的 frame(a.com/benign)以及一个可能被攻击的 frame(a.com/malicious),Web 应用开发者可以利用 HTML iframe 中的 sandbox 关键字来限制后一个 frame 的权限,如不允许弹窗或不允许提交表单等.然而,若该被攻击的 frame 具备执行 JavaScript 代码的权限,其可以访问良好的 frame 来实现弹窗或者提交表单的功能.Singh 等[13]分析 cookie 的访问控制包含 URL 路径,而同源策略忽略 URL 路径的策略实施机制将会破坏 cookie 的访问控制.Lerner 等[31]发现同源策略对同源不同 frame 的过度授权使得存档管理功能的 Web 应用受到了新的安全威胁.目前 Internet 上的最大存档网站(web.archive.org)将其他网站(如 http://www.xyz.com)保存

在诸如“https://web.archive.org/web/20001110101700/http://www.xyz.com/”的路径上.被存档的网站的源是作为URL 路径中的一个目录,这使得任意两个被存档的网站将被认为是来自于同一个源(https://web.archive.org/).如此,攻击者网页可以事先编写用来修改受害者网站内容的代码,在网页被存档后,该代码将修改受害者网页内容,从而影响诸如利用存档网站来进行法律取证的行为的正确性.

第二类是同源但在网页中处于不同位置的 frame.Son 等[14]通过分析目前网站在使用 postMessage API[57]上的不足,指出消息接收方需要使用基于 frame 的消息过滤方案.该方案通过消息监听函数参数 event 中的source 成员变量来完成更进一步的消息过滤,该变量是发送消息 frame 的引用,代表了其在该网页的 frame 结构中的位置.因此,即使该网页中存在另一个与消息发送方同源的其他 frame,该 frame 同样不能通过消息接收方的消息过滤方案.Barth 等[59]认为在一个网页中,一个 frame 只能对其子 frame 或者后代 frame 的 URL 赋值来重新加载 frame(浏览器中称这种行为为导航 Nagivate),其他兄弟 frame 是不具备导航权限的.然而,不论是基于 frame 的消息过滤方案还是导航机制都会因为同源策略的过度授权而被破坏.在同源策略的支撑下,一个不具备合法消息发送与导航权限的 frame 可以访问相应的被授权的同源 frame 来获取到对应的权限.

第三类是浏览器其他安全策略机制参与下的同源不同 frame.Stefan 等[60]依据强制访问控制机制设计了一种增强的安全策略 COWL,并被引入为 W3C 标准的一部分[61].Frame 需要处理其他源的 frame 的敏感数据,为了保证数据隐私,COWL 为处理数据的 frame 携带一个安全标签来表明其安全级别并对某些敏感操作(如发送网络请求)进行限制.同源的两个不同 frame 可能因为历史行为的不同具备不同的安全标签,并因此具备不同的访问控制权限.由于同源策略的过度授权,除非进行类似 frame 级别的安全隔离,COWL 的策略实施将会被破坏.Somé 等[10]发现浏览器的内容安全策略(Content Security Policy,简称 CSP)机制会被同源策略所破坏.内容安全策略允许 frame 的开发者通过设定 HTTP 头来指定 frame 的合法行为,如仅仅允许加载来自自身源的资源、不允许可能导致漏洞的 eval 操作等.通过对 Alexa top 10k 网站的内容安全策略协议进行分析,Somé 等发现在实施有内容安全策略的网站中,存在 72%的网页包含有同源或同域名的子 frame 并且父子 frame 的内容安全策略是不一致的.内容安全策略的不一致意味着两个 frame 被赋予了不同的权限,一个受限的 frame 同样能够借助授权的同源 frame 来进行权限提升.

3.2.2 防御方案

为应对同源不同 frame 的过度授权,研究者们从以下三个角度来设计安全防御机制.

第一类是细粒度的同源策略,即在同源策略基础上为主客体增加新的安全属性来区分同源不同 frame.特别的,不同 URL 路径的 frame 可以设置不同的允许访问列表,从而解决同源策略对同源不同 frame 的过度授权问题.Moshchuk 等[58]设计了细粒度的同源策略规则来实施基于内容的隔离策略.基于内容的隔离策略允许内容的所有者指定一个白名单.在浏览器环境中,Alice 可以将其网页 http://blog.com/alice/index.html 的白名单设置为Trust:list=http://blog.com/alice/*,安全监控器将认为该网页内容与来自于 http://blog.com/alice 目录下的网页是同一个主体,从而允许访问,但拒绝来自于 http://blog.com/网页的访问.除了可以实现基于 URL 路径的访问控制外,基于内容的隔离策略甚至能够使得两个不同源的网页被视作同一主体,这将有利于网页的资源共享需求.Jayaraman 等[62]允许 Web 服务器根据内容的可信度来为内容设置安全标签,并且将可信度不同的内容放置在不同环(ring)上.在策略实施时,同时确保低特权的环的主体不能访问高特权环的内容.Luo 等[63]认为基于环的方案没有处理不同主体(如同源不同 frame)在发送网络请求等行为上的不同,因此提出了基于能力(capability)的访问控制模型来加固同源策略.该访问控制模型中主体的安全属性是源与能力的集合.能力被定义为安全令牌(token).在策略实施时,确保只有具有特定的 token 的主体才能访问对应的资源.主体可以细粒度到 frame 甚至为 DOM 元素级别.Steven 等[64]、Cox 等[65]、Karlof 等[66]与 Dong 等[67]也在不同角度对同源策略进行细粒度策略增强来解决同源不同 frame 的过度授权问题.

第二类是动态的同源策略,即通过抛弃掉同源策略对源的依赖,允许 frame 更加灵活地设置主体或信任的资源,将能够可配置地解决同源策略对同源不同 frame 的过度授权问题.Cao 等[9]提出了一种可配置的源策略(Configurable Origin Policy,简称 COP)来替代同源策略.不同于同源策略以源作为访问控制主体的安全属

性,COP 是验证浏览器标注的可配置的 ID.一个 frame 可以自由更改它的 ID,并且确保 ID 是不可猜测的,以防恶意 frame 将自身主体安全属性设置为受害者的 ID.浏览器在进行访问控制时,只有主客体 ID 一致时才允许.若需要对同源不同 frame 进行隔离,授权的 frame 只需要设置新的 ID 并确保不将该 ID 分享给其他非授权的 frame 即可.同源策略对同源不同 frame 的过度授权将得以解决.此外,COP 还支持不同源之间的资源共享,这可以通过分享 ID 完成.

第三类是对浏览器其他安全机制的安全增强.Stefan 等[60]提出了 COWL 策略来防止敏感数据的泄漏,在该策略中,一个 frame 将根据接收到的来自其他 frame 的敏感数据来设置安全标签.考虑到同源策略对同源不同 frame 的过度授权,Stefan 等指出在浏览器进行同源策略验证时,将同时检查主客体的安全标签是否一致,若不一致则拒绝访问.这种两层的访问控制将不同安全标签的 frame 进行隔离,从而解决该场景下的同源不同 frame 过度授权问题.Somé 等[10]发现内容安全策略机制会被同源策略所破坏.从访问控制角度来说,这种漏洞存在的原因是同源策略的过度授权;从部署角度来说是因为同一网站中有不同 frame 具备不同的内容安全策略标签,因此可以从部署角度来减轻安全威胁.Pan 等[68]提出了 CSPAutoGen 系统.CSPAutoGen 系统通过训练的手段来为每个网站生成对应的模版,根据模版来为该网站设置内容安全策略规则.一个网站可以利用 CSPAutoGen 系统来为所有 frame 设置相同内容安全策略规则,从而避免由于同源策略导致的过度授权.此外,许多工作也对内容安全策略策略进行了分析[69−71]与不同角度的加固[72,73].

总的来说,前两类防御机制通过对同源策略进行修改来应对同源不同 frame 的过度授权问题.相比而言,第二类防御机制是从根本上解决问题,且除了实施细粒度的同源策略外,还可以将不同源的 frame 合并作为同一个主体.然而,对同源策略的安全增强为 Web 应用开发者提出了新的要求:Web 开发者需要确定资源的可信度而设置不同主体,这将影响这些新的策略的可行性.第三类机制通常针对于特定的安全威胁场景,没有从根本上来解决同源不同 frame 的过度授权问题,因此不具备普适性,但是不需要对同源策略进行更改从而保持向后兼容性.

3.3 不同源frame的过度授权与防御

浏览器同源策略的目标是进行源与源之间的隔离,然而,为了满足 Web 应用的需求,同源策略规则对两类特殊跨源资源访问没有作出限制:首先,同源策略允许一个 frame 向任意源的 Web 服务器发送数据;其次,同源策略允许一个 frame 使用<script src="xxx">的方式来执行任意源的脚本.研究发现,结合浏览器的 cookie 机制,攻击者可以利用这两类合法跨源资源请求来修改与偷取跨源数据,使得不同源的 frame 被过度授权.本小节将首先分析不同源 frame 被过度授权的场景及安全威胁,最后总结分析现有的安全防御工作.

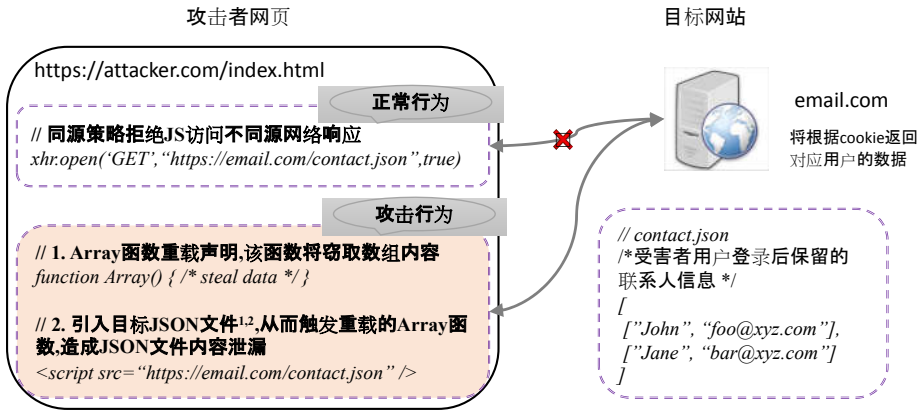
3.3.1 安全威胁

根据利用的同源策略规则的不同,研究者们发现了不同源 frame 可以获取过多权限的两种攻击:跨站请求伪造(cross-site request forgery,简称 CSRF)攻击与跨站脚本注入(cross-site script inclusion,简称 XSSI)攻击.

CSRF 攻击存在两种形式,包括 preAuth-CSRF (也被称为 Login CSRF) 攻击与 Auth-CSRF 攻击.preAuth-CSRF 攻击由 Barth 等[11]于 2008 年提出,该攻击发生在用户未登录受害者网站或者 cookie 已经过期的情况下.在 preAuth-CSRF 攻击中,用户由于无意或者受到钓鱼攻击等访问了攻击者网页.攻击者网站(attacker.com)与受害者网站(如 google.com)属于不同的源.当用户访问攻击者网页时,攻击者 frame 将主动向受害者网站发送一条 POST 网络请求,并且携带上攻击者自己选择的账号与密码.受害者网站将在认证账号密码后,将与用户浏览器协商 cookie 来保存会话.此后,用户可能在 cookie 还没有过期前访问了受害者网站的网页,并且执行一系列的操作,如在 Google 搜索引擎中进行搜索.由于 cookie 的存在,受害者网站服务器将会错误地认为该搜索行为来自于攻击者账号.因此,攻击者将能够事后在自己的浏览器上访问受害者网站来获取用户的历史搜索信息,从而突破同源策略对不同源 Web 应用资源的隔离限制.

与 preAuth-CSRF 攻击不同,Auth-CSRF 攻击发生在用户曾经登录过受害者网站并使得浏览器保存了对应 cookie 的情况下.在 Auth-CSRF 攻击中,用户由于无意或者受到钓鱼攻击等访问了攻击者网站(attacker.com),攻击者 frame 将自动向受害者网站(如 bank.com)的服务器发送一个转账请求.由于浏览器的 cookie 机制会自动为

网络请求携带上 cookie,而且用户事先登录过 bank.com 的 cookie 还未过期,bank.com 的服务器通过验证 cookie 将会认为转账请求来自于用户,从而在用户不知情情况下实现用户在 bank.com 上的转账. Sudhodanan 等[74]对 Auth-CSRF 攻击方式进行了总结,并对 Alexa 中的 300 个网站进行了检测,在 133 个可实施其实验的网站中, 90 个网站(68%)能够通过 Auth-CSRF 方式进行攻击.



[1] JSON文件在JavaScript语言中执行时为定义一个数组
[2] 浏览器在发送网络请求时将自动携带对应cookie,以便服务器认证用户

Fig 4 One form of XSSI attack through embedding remote JSON file[75]

图 4 利用 JSON 文件的 XSSI 攻击

XSSI 攻击来源于 Grossman 在 2006 年提出的 JSON 劫持攻击[75].图 4 描述了该攻击的场景与利用过程.具体来说, 用户事先登录过受害者网站(email.com),使得用户浏览器保存了对应的 cookie.在 cookie 没有过期时,用户被欺骗来访问攻击者网页(attacker.com).攻击者网页的目标是为了获取受害者网站的 JSON 文件(email.com/contacts.json),该 JSON 文件中包含了用户在受害者网站上的联系人信息.由于浏览器的同源策略,攻击者网页直接通过 JavaScript 的 API 来访问不同源 JSON 文件的行为会被拒绝.而在该 JSON 劫持攻击中,攻击者网页将利用同源策略允许的跨源内嵌脚本规则来引入受害者网站的 JSON 文件.受害者网站服务器通过对 cookie 的认证而返回用户的联系人信息.当浏览器接收到返回的 JSON 文件时,将其在攻击者网页中以脚本形式执行.由于 JSON 在 JavaScript 语言中被认为是定义一个数组,攻击者网页可以事先对数组(Array)进行重载,使得在构造数组时调用重载后的代码,该代码可以读取数组内容(即用户联系人),进而发送给攻击者服务器.最终,攻击者将能够获取用户在跨源服务器上的联系人信息.Grossman 向 Google 报告了这个问题,Christoph Kern 此后将该攻击命名为 XSSI 攻击,并在其 2007 年出版的书籍[76]中进行了详细描述.其他的一些变种 XSSI 攻击[77,78]也随之提出.

Lekies 等[12]发现 XSSI 攻击可以直接使用受害者网站的 JavaScript 脚本,而非 JSON 文件,这使得发起 XSSI 攻击更加便利与通用.Lekies 等认为目前 Web 应用在服务器端不再单一地使用静态脚本,相反,服务器会根据登录的用户往脚本中填入不同的信息(称之为动态脚本).通过对 150 个网站的实验分析,Lekies 等发现 49 个网站使用了动态脚本,其中携带的动态信息包含了用户的数据(如用户名与邮箱地址)、以及网页与 Web 应用通信的安全令牌等.与 JSON 劫持攻击类似,为获取这些信息,攻击者可以利用 HTML 的 script 标签将受害者网站的动态脚本引入到自身的 frame 中(图 4 中攻击行为的步骤 2).该动态脚本将在攻击者 frame 中执行.根据用户敏感信息在动态脚本中的不同定义方式(如全局变量、数组等),攻击者 frame 可以通过事先重载相应的 JavaScript 函数(图 4 中攻击行为的步骤 1)、或者事后访问全局变量来获取这些敏感信息.利用这些敏感信息,攻击者可以实现用户追踪或个性化的社会工程,更进一步的,Staicu 等[79]发现同源策略规则中允许嵌入的跨源图片也能被用来造成类似的隐私泄漏.

为了更直观地描述 CSRF 与 XSSI 攻击,我们总结了两种攻击所利用的同源策略规则、后果以及相关工作(见表 4).

Table 4 Attacks on accessing cross-origin resources (attacker and victim site are cross-origin)

表 4 获取跨源资源的攻击方式(攻击者网站与受害者网站不同源)

攻击		利用的同源策略规则	后果	相关工作
CSRF	preAuth-CSRF (login CSRF)	允许发送跨源网络请求	用户以攻击者账号浏览受害者网站,攻击者能**事后**获取用户浏览历史等信息	Barth 等[11]
	Auth-CSRF	允许发送跨源网络请求	攻击者以用户身份来**修改**受害者网站服务器(如转账)	Sudhodanan 等[74]
XSSI		允许发送跨源网络请求;允许嵌入跨源脚本(或图片)	攻击者以用户身份来**读取**用户在受害者网站服务器数据(如联系人)	Lekies 等[12]

注:CSRF 与 XSSI 分别为跨站请求伪造(cross-site request forgery)与跨站脚本注入(cross-site script inclusion)攻击的简称.

preAuth-CSRF、Auth-CSRF 与 XSSI 攻击的建立来自于两个不同的角度:

- 攻击时机:不同于 Auth-CSRF 与 XSSI 攻击,preAuth-CSRF 攻击发生在用户登录账号之前.不同的攻击时机使得攻击者采用不同的攻击方式,并且攻击获利方式也存在不同,如 preAuth-CSRF 需要攻击者事后登录自己账号来获取隐私数据;
- 攻击后果:攻击者网页利用 CSRF 攻击获取到的对不同源资源的访问权限是有限的.通常而言,CSRF 只允许攻击者获取用户历史行为或者修改受害者网站的状态,并不能允许攻击者直接读取用户的数据(如邮件网站中用户的联系人等信息).其本质原因在于同源策略规则允许一个 frame 向任意源网站发送请求,但不允许获取跨源网站返回的网络响应.XSSI 更进一步地考虑到同源策略规则允许 frame 内嵌任意源的脚本,利用这一规则将使得攻击者能够发起攻击来直接读取隐私数据.

3.3.2 防御方案

CSRF 与 XSSI 攻击得以发生的原因有两点:一是同源策略规则允许发送跨源网络请求与嵌入跨源脚本,二是浏览器 cookie 机制使得攻击者 frame 可以通过受害者网站服务器的用户认证.应对 CSRF 与 XSSI 攻击可以从这两个方面着手.

首先,同源策略规则的增强可以应对 CSRF 与 XSSI 攻击.Barth 等[11]指出浏览器为同源策略安全增加一个新的 HTTP 头:Referer.当一个 frame 发起网络请求时,浏览器将获取该 frame 的 URL,并将该 URL 作为 Referer 的值携带到网络请求中.尽管浏览器并未限制跨源网络请求,但是 Web 服务器可以根据 Referer 来进行同源检测,或着判断跨源网络请求是否来自于其信任的跨源网页.当发送 frame 不被认可时,Web 服务器将拒绝返回 XSSI 攻击所需的 JSON 与 JavaScript 文件、或者拒绝执行 CSRF 攻击中所要求的操作(如转账).尽管 Referer 对同源策略进行了加强,但是却被认为是导致信息泄漏的另一种方式[80,81],其原因在于一个网站可以通过 Referer 来判断用户通过其他哪些网站对其进行了访问,这将有助于建立网站排行与追踪用户行为,但破坏了用户隐私.

其次,对浏览器 cookie 机制的增强可以应对 CSRF 与 XSSI 攻击.Barth 等[11]认为可以通过安全令牌与 cookie 机制协作的方式来应对这些攻击.具体来说,一个 Web 应用可以在其所有 frame 的 URL 链接中加入安全令牌.当确保安全令牌很难被攻击者 frame 所获取时,Web 应用服务器可以通过安全令牌进行 cookie 验证后的第二次认证,用来判断网络请求是否来自于所允许的 frame.Web 应用同样可以在 frame 中使用 XMLHttpRequest API 来发送请求,并且同时携带上自定义的 HTTP 头与相应的值.当该自定义的 HTTP 头与值不能被攻击者 frame 所获取时,Web 应用服务器将同样能够区分发送网络请求的 frame 是否应该被授权.Franken 等[82,83]发现目前浏览器与浏览器扩展中实现了“阻止第三方 cookie”的功能.第三方 cookie 是指 frame 在网络请求中携带的跨源的 cookie,如 CSRF 与 XSSI 攻击中攻击者 frame 所利用的受害者网站的 cookie.浏览器自动阻止第三方 cookie 的

特性将能够应对 CSRF 与 XSSI 攻击.Franken 等设计一个框架来评估浏览器与浏览器扩展阻止第三方 cookie 功能的有效性,实验发现了现有实现中的几个缺陷,这些缺陷将导致阻止第三方 cookie 功能被绕过.

除了防御这些攻击外,研究者们还提出了一系列的方案来帮助 Web 应用检测其网页中是否存在 CSRF 攻击.Pellegrino 等[84]提出了一个基于模型的安全测试框架 Deemon 来自动检测 CSRF 漏洞.Deemon 能够动态追踪 Web 应用中诸如网络交互、服务器端执行与数据库操作等行为.根据追踪到的行为,Deemon 将推断出 Web 应用对应的模型图,然后利用推断的模型图进行图遍历来识别脆弱的网络请求.这些脆弱的网络请求将作为 CSRF 漏洞的候选,最后 Deemon 在真实的 Web 应用上对候选 CSRF 漏洞进行验证.实验发现了 10 个著名的开源 Web 应用上的 14 个未知的 CSRF 漏洞.Calzavara 等[85,86]提出了第一个基于机器学习的方法 Mitch 来进行 CSRF 漏洞检测.Mitch 将著名网站中的 5,828 个 HTTP 请求作为训练集来运行有监督的机器学习算法.实验表明 Mitch 在 20 个主要的网站中检测到了 35 个新的 CSRF 漏洞、以及 3 个已知形式但之前未检测到的 CSRF 漏洞.这些检测工具将有助于提醒 Web 应用开发者来进行安全加固.

通常而言,检测攻击的防御机制并不能从根本上防御这两类攻击,但能帮助 Web 应用开发者来进行安全加固.而通过加强同源策略规则与 cookie 机制的两类方案能应对 CSRF 与 XSSI 攻击,但带来了一些其他问题:前者在网络请求中携带了网页来源,从而面临着网页隐私泄漏风险,后者阻止第三方 cookie,但却对目前 Web 应用的用户追踪等其他需求造成了影响.

3.4 方案对比与讨论

同源策略规则不足主要体现在三个方面:同源策略允许第三方脚本以宿主 frame 的权限执行,攻击者因而可以利用第三方脚本来攻破宿主 frame,进而突破同源策略的限制(见 3.1 节);新型场景与策略的出现使得同源不同 frame 需要被赋予不同权限,但同源策略规则无法支持这些场景与策略(见 3.2 节);同源策略允许一个 frame 给任意源发送网络请求,导致攻击者可以利用其他浏览器机制来访问跨源资源(见 3.3 节).我们对这三类安全威胁进行总结分析,包括攻击方式、在当前 Internet 上发起攻击的影响范围评估及其代表性研究工作(见表 5).

Table 5 Comparison on security threates caused by limitations of same origin policy

表 5 同源策略规则不足所引起的安全威胁对比

	攻击方式	影响范围	代表性研究工作
第三方脚本的过度授权	劫持第三方脚本	Alexa top **10k** 网站中, **88.45%**的网站至少包含一个第三方脚本[6]	Nikiforakis 等[6]
	利用第三方脚本漏洞	Alexa top **75k** 网站中, **37%**的网站至少包含一个已知漏洞版本的第三方脚本[36]	Lauinger 等[36]; Stock 等[35]
	利用第三方脚本的间接引用	Alexa top **100k**网站中, **13%**的网站运行有隐式依赖的外部脚本[37]	Kumar 等[37]; Ikram 等[8]
同源不同 frame 的过度授权	利用不同 URL 的同源 frame	Alexa top **1M** 网站中,**80%**的网站将存在被攻击的风险[31]	Lerner 等[31]; Cao 等[9];
	利用不同位置的同源 frame	Alexa top **10k** 网站中,**22%**的网站注册了消息过滤器,存在被不同位置同源 frame 攻击的风险[14]	Son 等[14]; Barth 等[59]
	利用不同安全策略的同源 frame	Alexa top **10k** 网站的 1M 网页中,**31.1%**网页存在被攻击的风险[10]	Somé 等[10]; Stefan 等[60]
不同源 frame 的过度授权	CSRF 攻击	从 Alexa top 1500 选取的 **300** 个网站中,**68%**的网站能被攻击[74]	Sudhodanan 等[74]; Barth 等[11]
	XSSI 攻击	Alexa top **150** 网站中,**80%**的网站存在被攻击的风险[12]	Lekies 等[12]; Grossman [75]

表 6 对应对以上安全威胁的防御机制进行了总结分析,包括方案目标、网页加载开销、兼容性与可用性以及是否要求浏览器修改.总体而言,这些防御机制可以归为两个出发点:

- 增加强隔离机制:针对这些安全威胁,可以通过在同源策略的基础上增加强隔离机制来进行防御,包括应对第三方脚本过度授权的四类防御机制(见 3.1.3 节)、应对同源不同 frame 过度授权的前两类防御

机制(见 3.2.2 节)、以及应对不同源 frame 过度授权的同源策略增强机制(见 3.3.2 节).实现强隔离的优势在于普适性,即不需要对不同的场景与新型策略给出不同的解决方案.但强隔离机制的实现意味着需要设计合理且安全的资源共享机制,并将引入额外的开销.例如实现脚本之间的强隔离机制 TreeHouse 系统带来了接近 15 倍的网页加载开销[30];实现同源不同 frame 之间强隔离的 COWL 系统也带来了 16%的开销[60].这意味着实现强隔离的防御机制需要进一步优化,以保证隔离与资源共享的开销是在可接受的范围内;

- 实现权限控制:产生以上安全威胁的原因之一是新型场景与策略要求更细粒度的权限控制.例如,内容安全策略作用于 frame 上,从而导致同源不同 frame 被赋予了不同的权限[10],但 Web 应用开发者可以利用 CSPAutoGen[68]等系统来为同源的所有 frame 设置相同的内容安全策略规则,从而消除同源不同 frame 权限的不一致性.此类防御方案通常针对于特定的场景或者策略,并且对目前 Internet 上的网站可能带来兼容性问题,如 CSPAutoGen 系统对 Alexa top 50 网站中的 22 个网站的 UI 产生了细微的变化.然而,这一类机制不需要对同源策略规则进行变化,因而通常不需要对浏览器进行修改或需要网页开发者对编写习惯进行调整,并能对新型策略的设计与配置提供安全保障.

Table 6 Comparison on defenses against the limitations of same origin policy

表 6 针对同源策略规则不足的防御方案对比

防御方案	方案目标	网页加载开销	兼容性/可用性	浏览器是否需要修改
Jate[44]	实现脚本之间的强隔离	限制 Alexa top 500 网站的广告脚本:19.5%	脚本之间的函数调用必须通过代理对象完成	是
TreeHouse[30]	实现脚本之间的强隔离	限制基于 DOM 的游戏网页:15 倍	脚本之间的函数调用必须使用消息通信机制完成	否
AdJail[50]	实现脚本之间的强隔离	限制四种广告脚本:4.08%	脚本之间的函数调用必须使用消息通信机制完成	否
COP[9]	实现 frame 之间的强隔离	限制 Alexa top 200 网站:累计分布曲线在有无 COP 情况下基本一致	Alexa top 100 网站:完全兼容	是
COWL[60]	实现 frame 之间的强隔离	限制加密文档编辑器:16%	完全向后兼容	是
CSPAutoGen[68]	实现同源不同 frame 权限的一致性	限制 Alexa top 500 网站:9.1%	Alexa top 50 网站: 28 个网站兼容、22 个网站在 UI 上有极小区别	否
Barth 等[11]	利用 Referer 字段实现不同源之间的强隔离	-	在 HTTPS 协议上,仅有 0.05%-0.22%的浏览器取消 Referer 字段	是

3.5 小结

同源策略规则的设计在考虑安全性的同时需要满足 Web 应用所需的功能,因而允许了诸如允许发送跨源网络请求与嵌入第三方脚本等操作,从而造成同一个 frame 中第三方脚本的过度授权、同源不同 frame 的过度授权、甚至不同源 frame 的过度授权.这些安全威胁是策略设计对使用便利性的妥协,也意味着安全的浏览器环境必须要求策略设计者更合理的策略构建、以及 Web 应用开发者更安全的网页编写.Web 应用开发者不应盲目依靠同源策略来确保网站安全,还需要深入了解同源策略规则,并采取一系列的方案来应对一些同源策略规则所带来的不足.

4 跨域/跨源通信机制安全威胁及应对

浏览器根据同源策略来控制 JavaScript 代码对资源的访问.然而,随着 Web 的发展与 Web 应用功能的复杂化,越来越多的需要宽松的同源策略场景随之出现.这些场景可以分为两类:一是同父域名下跨源通信,二是不同域名下的跨域通信.在第一类场景下,一个域名的所有者需要搭建多种服务(如邮件、搜索以及新闻等),这些服

务分别搭建在不同的子域名下,但是有可能需要互相交互;在第二类场景下,一个网站希望借助于第三方的网站来完成某些特定任务,如用户追踪和性能分析等,该网站需要将某些数据分享给第三方网站.

为适应 Web 应用的新型场景,跨域/跨源通信机制随之而来.根据是否需要服务器端的参与,这些跨域/跨源通信机制可以分为两大类:无服务器辅助通信机制与服务器辅助通信机制.无服务器辅助通信机制依赖于特定的 JavaScript API,发送方通过调用 document.domain[87]修改自身身份来实现同域名下跨源通信,也可以通过调用 postMessage[57]来向跨域的接收方发送需要共享的消息.服务器辅助通信机制既可以依赖于特定的浏览器策略——跨域资源共享(CORS)[88],也可以由开发者利用一些 HTML 特性来自定义实现,如 JSON-P[89].图 5 总结了两大类跨域/跨源通信机制的发展过程,本节将重点对这两类机制的现有研究工作进行总结分析.

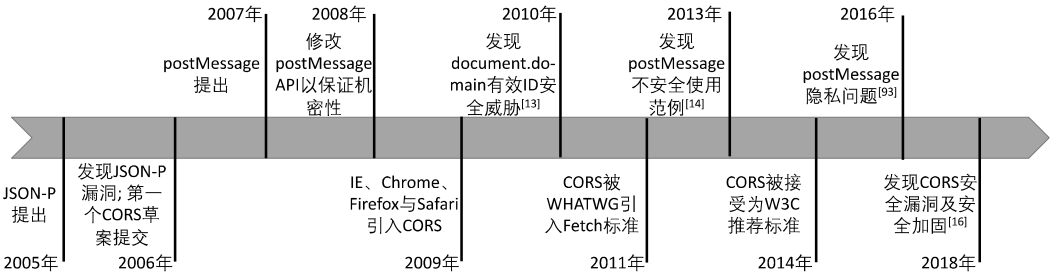


Fig 5 Cross-domain/cross-origin communication mechanisms

图 5 跨域/跨源通信机制

4.1 无服务器辅助通信机制威胁与应对

4.1.1 document.domain 跨域通信

(1). 机制概述

网页可以通过对 document.domain 的赋值来将自身提升为父域名.举例来说,一个网页的 URL 为 http://mail.xyz.com:8080/index.html,则该网页的源为三元组<http, mail.xyz.com, 8080>,当该网页执行了 document.domain="xyz.com"后,浏览器将该网页的源更新为三元组<http, xyz.com, null>,注意源中的端口字段被设置为空(null).当目标网页来自于父域名(如 http://xyz.com:8080/)或者兄弟域名(如 http://video.xyz.com:8080/),并且执行了 document.domain="xyz.com"时,目标网页的源更新为三元组<http, xyz.com, null>.此时,浏览器的同源策略将本网页与目标网页视为同源,从而允许资源访问.特别地,一个网页不能设置 document.domain 为顶级域名,如.com、.org 等,且<http, xyz.com, 8080>与<http, xyz.com, null>不同源.

(2). 安全威胁与应对

使用 document.domain 进行跨源通信的安全威胁是同源策略在处理资源访问上的不一致性.Singh 等人[13]阐述了这种不一致性,并据此构造了多种攻击来绕过同源策略的限制.一个源的资源包含了 DOM 和 JavaScript 对象、cookie、本地存储以及用来发起网络请求与获取网络响应的 XMLHttpRequest API.当一个网页使用 document.domain 进行域名提升后,我们称提升后的源为该网页的有效标识(effective ID).在域名提升之后,当该网页访问其他网页的 DOM 和 JavaScript 对象时,同源策略将使用有效标识进行决策.但是,对于 cookie 和本地存储的访问以及 XMLHttpRequest 的使用,同源策略仍然使用该网页初始的源作为访问主体标识.基于同源策略在检查资源访问时处理有效标识的不一致性,攻击者网页能够构造攻击来偷取受害者网页的 cookie 与本地存储、获取受害者网站服务器数据(如获取用户邮箱等)或更改受害者网站服务器状态(如转账等操作).

图 6 给出了通过利用这种不一致性来获取跨源 cookie 与网络资源的攻击.在这一类攻击中,受害者网站(x.a.com)上有一个网页(1.html)使用 document.domain 将其有效标识设置为 a.com,这种域名提升将允许 1.html 访问 a.com 的某些资源.然而,当攻击者拥有来自于 y.a.com 的网页 attacker.html 时,攻击者将具备发起此类攻击的条件.具体攻击过程如下:

(1) attacker.html 向 1.html 注入脚本.由于 attacker.html 与 1.html 的有效标识一致,同源策略将允许 attacker.html 访问 1.html 的 DOM 对象,从而实现恶意脚本的注入;

(2) 注入到 1.html 中的恶意脚本发起攻击.如图 6(a)所示,恶意脚本可以访问与 1.html 同源的 2.html,从而可以借助 2.html 获取 x.a.com 的 cookie,并返回给 attacker.html;如图 6(b)所示,恶意脚本可以直接调用 XMLHttpRequest API 向 x.a.com 服务器发送恶意网络请求,并成功获取服务器返回的网络响应.

(a) 获取跨源 cookie (b) 发送恶意网络请求

Fig 6 Two attacks by using the inconsistency of same origin policy on document.domain[13]

图 6 利用同源策略对 document.domain 机制的不一致性发起的两种攻击[13]

Singh 等人[13]认为解决这种攻击的措施是让同源策略在处理 cookie 与 XMLHttpRequest 等 API 时也使用有效标识.然而,采用这种措施可能会造成其他的安全问题.例如,来自于 y.a.com 的恶意网页 attacker.html 在提升域名为 a.com 后,将能够获取 a.com 的 cookie.考虑到这些安全问题的存在,到目前为止仍然没有被广泛接受的防御方案,而目前的浏览器依然受到这一系列攻击的影响.值得庆幸的是,发起这类攻击需要攻击者拥有一个与受害者网页的兄弟域名,这在一定程度上增加了攻击者发起攻击的难度与成本.但不论如何,研究一种不影响可用性且能防御该类攻击的方案是很有必要的.

4.1.2 postMessage 跨源通信

(1). 机制概述

基于 document.domain 的通信机制要求通信双方网页具备相同的域名后缀,而基于 postMessage 的通信机制没有这一限制,从而允许向任意源的网页发送数据.由于网页大量地内嵌第三方网页用作用户追踪、性能分析等,postMessage 得以广泛使用[90].

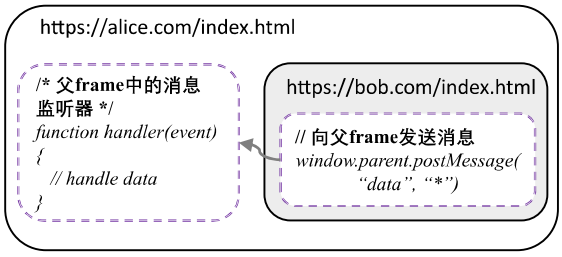


Fig 7 Example usage for postMessage API

图 7 postMessage 跨源机制使用示例

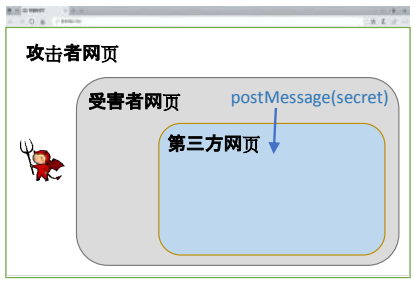
图 7 给出了 postMessage API 的使用示例.在该示例中,来自 alice.com 的 frame A 使用 iframe 标签内嵌了来自 bob.com 的子 frame B,因此 frame B 的 window.parent 将指向 frame A 的 window 对象.Frame B 执行了 window.parent.postMessage 来向 frame A(由 window.parent 指定)发送跨源消息.postMessage 的第一个参数为消息内容,第二个参数为预期接收者的源.当 frame A 收到消息时,frame A 中事先注册的消息处理函数 handler(可自定义)将被调用,其参数为 event,包含三个成员:event.source(发送方的 window 对象)、event.data(消息内容)、

event.origin(发送方的源).Frame A 可以使用 event.source.postMessage 向 frame B 返回消息.

(2). 安全威胁与应对

postMessage 在 2007 年被提出时仅仅提供了一个参数来传输要发送的消息内容,Barth 等[59]发现了 postMessage 无法满足机密性要求.如图 8(a)所示,受害者网页内嵌了一个第三方网页并且使用 postMessage 来传输数据,该数据可能是用户的隐私数据.例如,受害者网页可能借助第三方网页来验证用户口令强度,此时的用户口令将使用 postMessage 的方式传输给第三方网页.在这种情况下,攻击者诱导用户访问其网页,该网页内嵌了受害者网页,进而内嵌了第三方网页.由于第三方网页为攻击者网页的孙子网页,攻击者网页可以对第三方网页进行导航(修改 window.location.href 变量),导致浏览器将第三方网页重新加载为攻击者网页(如图 8(b)所示).此时,受害者网页通过 postMessage 发送的秘密数据将被重新加载后的攻击者网页所接收,导致秘密数据泄漏.

这种针对 postMessage 机密性的攻击成功的原因在于浏览器未考虑第三方网页被重新加载为攻击者网页.Barth 等[59]建议 postMessage API 增加第二个参数,用来表明发送方预期的接收方的源.当浏览器将受害者发送的秘密数据提交给第三方网页时,浏览器判断第三方网页是否仍然属于原来的源.在图 8 的攻击中,由于第三方网页被重新加载,其源变化为攻击者网页,导致浏览器对源的判断失败,浏览器将拒绝向该攻击者网页转发原来的消息,从而攻击者无法获取到受害者网页的秘密数据.HTML 标准据此对 postMessage API 进行了修改而引入了第二个参数,目前浏览器已经能够抵抗图 8 所示的攻击.



(a) 受害者网页往第三方网页发送 secret

(b) 攻击者替换第三方网页来偷取 secret

Fig 8 Attacks on the confidentiality of postMessage API

图 8 针对 postMessage 机密性的攻击

在此之后,Son 等[14]发现了使用 postMessage API 的另一个安全威胁.postMessage 涉及到消息发送方与接收方,接收方需要验证消息是否来自于合法的发送方.否则,恶意的消息发送方(攻击者网页)能够通过消息来在接收方(受害者网页)实现数据注入攻击.例如,当接收方网页将消息中的某些字段写入 cookie 时,攻击者可以通过消息来篡改接收方网页的 cookie 内容;当接收方网页将消息中一些字段利用 JavaScript 提供的 eval 等 API 来当成代码执行时,攻击者可以在受害者网页中执行任意代码.HTML5 标准中明确指出,消息接收方需要判断发送方的源[91].然而,Son 等[14]对现有 Alexa top 10000 网页进行了调研分析,发现现有网页中存在不足:

- 2245 个网站上使用了 postMessage API;1585 个网站没有对发送方进行验证;
- 261 个网站对发送方的源进行了验证,但是验证方式是不安全的.例如有网站对发送方判断的代码为 if(m.origin.indexOf("sharethis.com") != -1),这使得任意在源中含有 sharethis.com 的网页发送的消息都会被接收方所接受,此时,攻击者可以通过注册类似于 sharethis.com.malicious.com 或者 evilsharethis.com 等域名来发起攻击;
- 在未对发送方源进行验证或进行不安全验证的网站中,84 个网站允许攻击者网页在接收方网页执行任意代码或者往本地存储中注入任意内容.

针对这一类安全威胁,Son 等[14]提出了两种防御方式.第一种方式要求发送方与接收方事先构造某个秘密数据,由于同源策略限制,使得只有与发送方具备相同源的网页能够访问这个秘密数据,接收方消息处理函数可

以通过验证该秘密数据来判断发送方是否为授权方;第二种方式要求接收方组合 event.origin 与 event.source 来验证发送方身份,合理地组合这两个对象能够实现接收方只接受某个特定源的特定网页发送的消息.Weissbacher 等[92]提出了一个检测这类安全威胁的系统 ZigZag.通过透明地在网页代码中插桩,ZigZag 能够在浏览器执行过程中实时地执行不变量检测.从这些不变量中,ZigZag 推断出网页代码的正常行为模型,这些模型捕获客户端 Web 应用程序组件如何交互(以及与谁交互)的基本属性、以及与浏览器中的控制流和数据值相关的属性.使用这些模型,ZigZag 可以自动检测与这类安全威胁高度相关的模型的偏差.

此外,浏览器的事件机制使得 postMessage API 存在隐私泄漏的风险.Guan 等[93,94]针对于这类安全威胁给出了具体的攻击场景与攻击方案,并且提供了对应的抵抗措施.Guan 等首先分析了目前的网页中一个普遍的使用范例.当网页 A 内嵌第三方网页 B 并使用 postMessage API 进行通信时,目前网页开发者倾向于在网页 A 中引入 B 的一个脚本当作第三方库,这个第三方库将在网页 A 中注册消息处理函数来完成诸如消息格式解析等任务.据此,网页开发者不需要了解网页 B 的消息通信机制,使得开发更加便利.然而,当网页 A 由于功能需求需要内嵌多个第三方网页(如 B 和 C)时,B 和 C 都将在网页 A 中注册消息处理函数.当网页 A 收到消息时,根据浏览器的事件机制,所有在网页 A 中注册的消息处理函数都会被触发,这意味着网页 B 向 A 发送的消息将能够被 C 所监听.当消息中存在秘密数据如 cookie 时,用户在 B 网站上的隐私将可能会被 C 所破坏.Guan 等人[93]将这类攻击称之为 DangerNeighbor 攻击,他们还对目前 top 5000 的网站进行了调研分析,发现 28.8%的网站注册了消息处理函数,10.88%的网站注册了来自不同源的多个处理函数,因而可以认为 10.88%的网站可能受到 DangerNeighbor 攻击的影响.为抵抗 DangerNeighbor 攻击,发送方需要发送加密数据而非消息明文;接收方利用 JavaScript 函数的特性,封装注册消息处理函数,同时封装的处理函数负责调用相应的原处理函数,而其他处理函数不被调用,从而解决 DangerNeighbor 攻击所带来的隐私泄漏威胁.

4.2 服务器辅助通信机制威胁与应对

4.2.1 JSON-P 跨源通信

(1). 机制概述

基于 JSON-P 的跨源通信机制的基础是同源策略允许内联第三方脚本.如图 9 所示,来自于 alice.com 的网页可以借助<script>来内联一个跨域的网站 bob.com 的脚本 data.js,虽然该网页无法读取 data.js 的内容,但是同源策略允许 data.js 在该网页中执行.此时,若 bob.com 希望把自己的某些数据共享,其可以在 data.js 中调用一个 my_callback 函数,其参数为需要共享的数据,并且以 JSON 的形式存在.当 alice.com 的网页事先定义了 my_callback 函数时,data.js 将调用该预先定义的函数,从而网页可以获取到来自跨源的 bob.com 的数据,实现跨源或跨域资源共享.特别注意的是,JSON-P 只能支持 GET 请求,不能支持 POST 等其他请求.

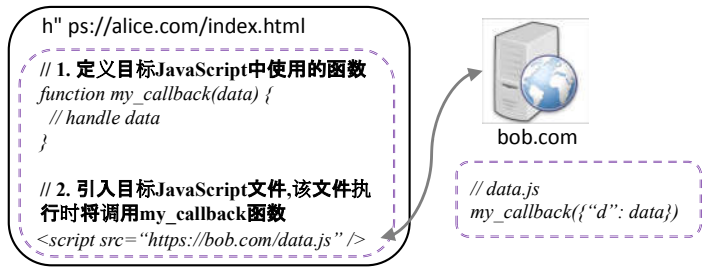


Fig 9 Example usage for JSON-P

图 9 JSON-P 跨源机制的使用示例

(2). 安全威胁

基于 JSON-P 实现的跨域机制依赖于同源策略对引入跨域脚本的允许以及服务器端的辅助,其面对的一个最大的安全威胁是如何防止攻击者网页(如 https://alice.com/index.html)访问服务器想要共享的 JSON-P 资源(如

https://bob.com/data.js).Popescu[15]按照服务器端共享资源的方式,总结了恶意获取 JSON-P 资源的几种途径:

- 如果 data.js 中调用了特定函数来返回共享资源(my_callback({'data':data})),攻击者网页可以通过先定义 my_callback 函数再内联 data.js 来恶意获取资源;
- 如果 data.js 中调用了对象成员函数来返回共享资源(System.TransactionData.Fetch({'data':data})),攻击者可以通过事先依次定义对象(System.TransactionData)与对象成员 Fetch 函数来恶意获取资源;
- 如果 data.js 允许通过 URL 来指定回调函数,攻击者可以通过自定义函数来恶意获取资源.例如当 URL 为 https://bob.com/data.js?callback=my_callback 时,data.js 将调用 my_callback 函数,此时攻击者网页可以通过定义 my_callback 函数,再内联以上 URL 的脚本来获取资源;
- 如果 data.js 利用 URL 指定的字段并加入数字来指定回调函数,攻击者可以通过先遍历所有可能的回调函数名称并定义对应函数来获取资源.

最后攻击者网页可以通过网络通信来将获取的恶意数据返回给攻击者服务器.此外,Grossman[75]发现可以攻击使用 JSON-P 机制的 Google 邮件网站.当用户事先登录了 Google 邮件网站后,浏览器中维护了用户对应的 cookie.当用户 cookie 没有过期的情况下,用户被诱导访问了攻击者网页,该攻击者网页嵌入了 Google 邮件网站的脚本(如 http://mail.google.com/mail/?_url_scrubbed_ 来获取联系人列表).由于浏览器在发送网络请求时会自动携带对应的 cookie,Google 邮件服务器将会返回受害者用户的联系人列表.联系人列表在 JavaScript 中将会转为数组 Array,攻击者网页可以通过事先重载 Array 对象来读取联系人列表.这种攻击是 XSSI 攻击(见 3.1.1 节)的一个例子,JSON-P 机制为发起 XSSI 攻击创造了条件.

4.2.2 CORS 跨域资源共享

(1). 机制概述

Web 的 Fetch 标准[95]定义了 CORS 机制的基本规范来允许一个 frame 获取跨源服务器的网络响应.CORS 中定义了两类请求:简单请求与非简单请求.前者要求请求方法为 HEAD、GET 或 POST,且 HTTP 头信息为预先定义的几种字段(如 Accept、Content-Language 等).后者采用诸如 PUT、DELETE 之类的请求方法或使用一些其他的 HTTP 头字段.非简单请求被认为是可能存在安全风险的网络请求.

对于简单请求,CORS 要求浏览器在发送网络请求时增加名为 ORIGIN 的 HTTP 头,用来说明请求网页的源;若目标服务器允许该源访问,则在网络响应中增加 Access-Control-Allow-Origin 的 HTTP 头,值为服务器允许访问网络响应的源;浏览器在收到网络响应后,仅当请求网页的源被包含在 Access-Control-Allow-Origin 中时,才允许请求网页访问网络响应.

非简单请求需要事先执行预检协议(preflight).在预检协议中,浏览器采用 OPTIONS 请求方法,其中包含了 Access-Control-Request-Method 来指定跨源请求的请求方法(如 PUT)、以及 Access-Control-Request-Headers 来表示跨源请求中特殊或者自定义的 HTTP 头字段.服务器若允许该源访问资源,将返回 Access-Control-Allow-Origin、Access-Control-Allow-Methods 与 Access-Control-Allow-Headers 来表示允许资源共享的源、请求方法以及 HTTP 头;当浏览器发现服务器允许时,将执行简单请求对应的协议来让网页访问跨源服务器数据.此外,为了提高性能,CORS 机制还提供了 Access-Control-Max-Age 头字段来让浏览器缓存预检协议的结果.

(2). 安全威胁

Chen 等[16]对 CORS 在现实世界网页中的实际应用进行了实证研究.通过对 CORS 的设计、实现和部署的分析,他们发现 CORS 受到许多新的安全问题的影响.考虑到这些安全风险,Chen 等提出了一些建议来对协议进行简化和澄清,包括对简单请求也进行对应的预检、对简单请求携带的 HTTP 头与内容的格式与编码等进行额外的限制等.其中一些建议已经被 CORS 规范和主流浏览器采用.

CORS 机制的第一类安全风险是 CORS 协议实现放宽了跨源"写"特权.CORS 对于简单请求没有预检阶段.对于简单请求中允许的 9 个 HTTP 头字段,RFC7231 中明确指出其中的 4 个字段有格式的限制(如 Accept、Accept-Language、Content-Language 以及 Content-Type).然而目前主流浏览器中只有 Safari 进行了格式限制,

而且所有主流的浏览器对 Content-Type 字段的限制都是基于前缀匹配,这意味着攻击者网页在简单请求中携带恶意负载,服务器对 HTTP 头信息不安全的解析将造成安全威胁.Chen 等证明了不安全的格式检查能够使得攻击者网页在 Apache 服务器端远程执行代码.此外,CORS 没有对简单请求 HTTP 头信息的总字段进行限制,而浏览器通常对 HTTP 请求长度存在限制,超过长度限制将返回 HTTP 400 错误,否则正常返回 HTTP 200.这种不一致性可以允许攻击者网页通过构造 CORS 请求来推断受害者网页是否存在 cookie 等信息.推断 cookie 的存在能够被用来偷取受害者用户的隐私,如用户登录了医院网站可以推断出用户可能存在某些病症.另外,CORS 请求对 HTTP 内容也没有格式或者编码的限制,这也能造成安全风险.

CORS 机制的第二类安全风险是其为网络交互引入了新的风险信任依赖.CORS 机制使得一个网站能够访问跨源服务器的资源,导致跨源服务器与请求网站存在信任依赖.如此,当攻击者有机会攻破跨源服务器某个允许访问的网站时,跨源服务器的资源将会被泄漏,这种情况在 HTTPS 服务器信任 HTTP 同网站网页时最为明显.

CORS 机制的第三类安全风险是其配置通常不被开发人员很好地理解.由于它的无表达策略和它与其他 Web 机制的复杂的交互,CORS 协议配置出现了各种各样的错误.开发人员可能按照前缀、后缀或正则等匹配方式来确定允许访问服务器资源的源,这种匹配方式可能存在漏洞,使得攻击者可以构造满足匹配的网址域名来恶意偷取跨源服务器的资源.

4.3 方案对比与讨论

无服务器辅助的两类通信机制分别基于 document.domain 与 postMessage API,我们对这两种机制进行总结分析,包括通信范围、攻击后果、发起攻击的影响范围评估以及其在目前网站中的使用率(见表 7).

Table 7 Comparison on mechanisms of cross-domain or cross-origin data sharing without server's intervention

表 7 无服务器辅助的跨域/跨源通信机制对比

通信机制	通信范围	安全威胁	影响范围	在目前网站中的使用率*
document.domain	同一父域名下的兄弟域名、或同一网站的祖先与子孙域名	利用资源访问上存在不一致性,攻击者可访问受害者源范围内的任意资源[13]	**Alexa top 100k** 网站中,**1.87%** 的网站访问了不一致授权的资源	7.4%
postMessage	任意源	攻击者可以将接收方 frame 导航为自身 frame 以偷取消息内容(目前机制已修复) [59]	-	49.5%
		依赖于消息接收方的处理逻辑,可以造成脚本注入或 cookie 重写等后果[14]	**Alexa top 10k** 网站中,2245 个网站(**2.25%**)注册了消息监听器, 1845 个网站(**1.85%**)存在被攻击的风险	
		第三方脚本提供商能监听网页消息内容以偷取隐私数据[93]	**Alexa top 5k** 网站中, 1440 个网站(**28.8%**)注册了消息监听器, 544 个网站(**10.88%**)的网站存在被攻击的风险	

*数据来自于 Chrome 信息统计网站(https://chromestatus.com/metrics/feature/popularity),数据采集时间为 2020 年 3 月 24 日.

其中,document.domain 局限于同一网站内的资源共享,而且共享成功的两个 frame 具有任意资源访问权限.尽管有研究者们提出了基于 document.domain 机制的安全的跨源通信通道[96],但是 Web 应用仍越来越倾向于使用 postMessage API.然而,postMessage API 的使用范例使得 Web 应用存在数据注入攻击与隐私泄漏的问题,以至于 1.85%至 10.88%的使用 postMessage 机制通信的网站存在被攻击的风险[14,93].现有的解决方案的思路在于对网页代码进行重写以建立正常行为模型或阻止其他处理函数的调用.前者存在误报与影响正常跨源通信的风险,而后者由于处理函数具备与宿主 frame 相同权限而存在绕过网页重写防御机制的缺陷.研究者们需要在这一方面进行更进一步的研究以提出更合理、完善的解决方案.

在服务器辅助的跨源通信机制上,CORS 已经被引入至 HTML5 标准中,而 JSON-P 是开发者对于跨源需求所提出的方案,因此其没有统一的规范来实施以解决面临的安全问题.此外,基于 JSON-P 实现的跨域机制还面临着信任问题.因为 JSON-P 资源是作为 JavaScript 立即执行的,引入该资源脚本的网页不能对 JSON-P 资源执行任何输入验证.若 JSON-P 资源含有恶意代码,引入该资源脚本的网页将受到攻击.因此,导入第三方 JSON-P 资源需要完全信任第三方.Stock 等[90]曾对目前网站中 JSON-P 与 CORS 的使用率进行了统计,发现 2014 年前,JSON-P 的使用率远大于 CORS,然而截至 2016 年,CORS 的使用率已经超过 JSON-P 将近 10%.CORS 机制相对于更加完善,HTML5 标准的支持也将不断修复 CORS 机制中存在的安全漏洞,因此 Web 应用开发者将逐渐趋向于使用 CORS 来取代 JSON-P.

4.4 小结

目前的跨域/跨源通信方案最主要的四种机制是 document.domain、postMessage、JSON-P 与 CORS. document.domain 只支持同网站下不同子域名之间的资源共享,而 JSON-P 由于其在可用性与安全性上的不足已经基本被 CORS 所替代.之后的网站将越来越倾向于使用 postMessage 与 CORS 这两类标准中提供的跨源通信机制.然而,postMessage 与 CORS 依旧存在各种各样的安全威胁,研究人员在不断地分析安全威胁与提出更加安全的机制,网站开发人员应当了解使用这些机制的风险并尽可能按照标准建议的方案进行代码编写.不论如何,跨域/跨源通信机制允许不同源网站共享资源,这意味着网站的信任边界从本网站自身向外扩大,我们要认识到信任边界扩大的安全威胁,不应为了便利而盲目地使用这些通信机制.

5 内存攻击下的同源策略安全

使用不安全语言(如 C/C++)开发的软件不可避免地存在引起内存攻击的漏洞,如缓冲区溢出、内存泄漏与格式化字符串等[97].内存攻击的初始形式为代码注入攻击[98],即攻击者在程序运行时利用程序的漏洞注入事先准备的负载(payload),然后通过将程序控制流指向负载,导致负载作为恶意代码得以执行.应对代码注入攻击的防御机制是数据不可执行(Data Execution Prevention,简称 DEP)机制[99].DEP 通过将数据段与代码段分离,并确保数据段不可执行.此后,代码重用攻击得以提出,例如 ROP(Return Oriented Programming)[100,101]、JOP(Jump Oriented Programming)[102,103]等.代码重用攻击不需要注入代码,而是从现有程序代码中找到一系列的片段(称为 gadgets)并修改栈来链接 gadgets.代码重用攻击被证明能执行任意代码,但控制流完整性(Control Flow Integrity,简称 CFI)[104,105]以及一些基于特征的启发式方式[106,107]能够被用来检测与防御代码重用攻击.其他抵御代码重用攻击的方案还包括修改编译器来使得程序没有可利用的 gadgets[108],或者通过地址空间随机化[109]来使得攻击者很难找到 gadgets.更进一步的,内存攻击还能允许攻击者直接对内存中的数据进行读取或修改(称之为仅数据攻击).基于数据流的系统化的攻击(Data Oriented Programming,简称 DOP)[110]与防御机制(Data Flow Integrity,简称 DFI)[111]也随后得以提出.

浏览器环境的特殊性使得内存攻防研究进入一个新的阶段.

- 代码注入攻击:浏览器通过引入在运行 JavaScript 时编译 JavaScript 的即时编译器(Just-In-Time Compiler,简称 JIT 编译器)来提高性能,这使得 JIT 编译后生成的代码(称为 JIT 代码)所在内存页的权限必然是变化的:在生成 JIT 代码时内存页的权限为可写,运行 JIT 代码时内存页的权限为可读.这意味着攻击者有可能来发起代码注入攻击;
- 代码重用攻击:JIT 编译意味着一个恶意的 frame 可以通过设计相应的 JavaScript 代码来在浏览器中注入代码重用攻击所需要的 gadgets,从而传统的使用特定编译器来去除 gadgets 的方案并不能抵御对浏览器的代码重用攻击.而且,浏览器的性能需求使得利用 CFI 来加固 JIT 代码是不可行的;
- 仅数据攻击:一个网页中可能包含来自不同源的 frame,这些不同源的代码将在同一进程中执行.一个恶意 frame 可以利用内存攻击来直接修改受害者 frame 的数据.

浏览器环境中的内存攻击将对同源策略安全造成破坏性的影响.成功发起代码注入攻击和代码重用攻击意味着攻击者 frame 可以直接执行任意代码,同源策略的安全防御能够很容易被绕过.发起仅数据攻击可以直

接读取与修改受害者 frame 的数据.由于同源策略的安全监控器与网页代码位于同一个渲染进程里,仅数据攻击也可以通过修改浏览器的元数据来欺骗同源策略的安全监控器.考虑到浏览器(特别是 JIT 编译器)漏洞的不可避免性,研究分析可行的内存攻击以及设计合适的防御方案是浏览器同源策略安全研究中不可缺少的环节.本节将对现有浏览器内存攻防研究工作进行总结分析.

5.1 代码注入攻击及防御

5.1.1 攻击方案

针对浏览器的代码注入攻击通常利用了 JIT 编译器的特性.首先,JIT 编译器将在运行时将 JIT 代码写入内存页中,这意味着 JIT 代码页在某些时候需要设置为可写.Gong 等[18]发现 Chrome 中的 JIT 代码页是同时具备写与可执行权限的,据此实现了代码注入攻击.具体来说,Gong 等首先利用 Chrome 的 V8 JavaScript 引擎中的漏洞来获取对任意内存地址的读写权限,其后分析 Chrome 代码来追踪到 JIT 代码的入口地址,最后通过对该地址的内存页的重写来注入恶意代码.更进一步的,Song 等[17]认为应对代码注入攻击的传统 W^X(不能同时可写与可执行)方式在浏览器环境中是无用的.W^X 通过限制内存页的权限不能同时为可写与可执行来抵抗代码注入攻击,然而,浏览器中 JIT 编译器产生新代码或优化现有 JIT 代码时,必须将 JIT 代码页短时间内更改为可写,这个短暂的时间窗口可以被敌手利用来将恶意代码注入到 JIT 内存页中.

其次,JIT 编译器生成 JIT 代码的一些规则也能用来发起代码注入攻击.Blazakis 等[19]提出的 JIT Spraying 攻击能够利用 JIT 编译器忽略大常数的特性来在 JIT 代码中注入所需的代码.图 10 展示了 JIT Spraying 攻击的原理.具体来说,攻击者 frame 的 JavaScript 代码可以定义一个含有大常数的变量,如 a=(0x11223344^0x44332211^0x44332211).该代码被 JIT 编译器编译后的二进制代码在图 10 中进行了标注,其中对变量 a 赋值的大常数直接存在于 JIT 代码中.若攻击者 frame 利用控制流劫持漏洞使得程序跳转到 JIT 代码的第二个字节执行,此后浏览器执行的程序逻辑将被篡改.所以,攻击者 frame 可以通过合理地组织 JavaScript 大常数来在 JIT 代码中注入恶意代码.为增大控制流跳转到注入代码特定地址的概率,JIT Spraying 攻击要求攻击者 frame 重复地注入大量的代码页,这一操作被称作为 Spraying.

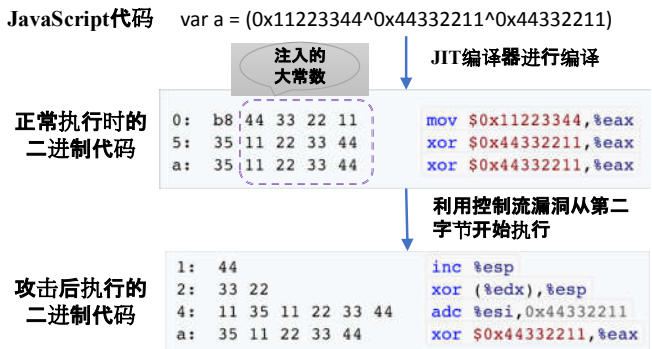


Fig 10 Example of injecting code into browsers by JIT Spraying attack

图 10 JIT Spraying 注入代码的示例

5.1.2 防御方案

与传统环境中的应对代码注入攻击方案相同,浏览器环境中也首先需要对 JIT 代码页采用 W^X 方案.Chen 等[112]提出了 JITDefender 系统来在 JIT 编译器中增加 W^X 机制.JITDefender 机制能够标识 JIT 代码的两个时间点:JIT 编译器生成 JIT 代码与 JIT 代码开始被执行.JITDefender 的工作流是在第一个时间点时将 JIT 代码页权限设置为可写但不可执行,在第二个时间点将 JIT 代码页设置为可执行但不可写.Crane 等[113]设计 Readactor 架构来实现仅可执行内存页.Readactor 设计了对应的编译器来将未修改的源代码进行转换,其中实现了分离代码和数据来消除对代码页的读访问、随机化代码布局与设计跳板代码(trampoline)来隐藏代码指针等操作.基于

该编译器,一个应用程序的代码页将不需要有读写权限.Readactor 之后利用 Intel 处理器所提供的扩展页表(Extended Page Tables,简称 EPT)来实现硬件限制的仅可执行内存页.目前主流浏览器已经实施了对 JIT 代码的 W^X [114]保护.

尽管 JITDefender 与 Readactor 在浏览器中实现了 W^X,但是 JIT 编译器在生成 JIT 代码与执行 JIT 代码之间仍然有短暂的时间窗口来发起代码注入攻击.基于此,Song 等[17]建议将 JIT 引擎拆分为两个不同的进程来应对这种安全威胁,包括一个执行 JIT 代码的不受信任进程,和一个发出 JIT 代码的受信任进程.这种体系结构可以防止 JIT 内存在不受信任的进程中随时可写.由于分割 JIT 引擎需要进程间通信和消息同步,因此在 JavaScript 基准测试中该方案所引入的运行时开销高达 50%.此外,Chen 等[115]设计了 JITSafe 框架来混淆 JIT 代码与缩小 JIT 编译器生成与执行 JIT 代码的时间窗口.Frassetto 等[116]利用地址空间随机化技术来使得攻击者无法获取 JIT 代码页的地址,从而抵抗利用该时间窗口的代码注入攻击.

利用 JIT 编译器忽略处理大常数的特性来发起的 JIT Sparying 攻击能够更进一步地加强为代码重用攻击,因此将在下一节中具体阐述其加强后的攻击与防御方案.

5.2 代码重用攻击及防御

5.2.1 攻击方案

代码重用攻击是通过链接小的代码指令片段(gadgets)来执行任意代码,因而需要两个前提:一是目标程序中具备有足够的 gadgets,二是目标程序存在控制流劫持漏洞使得攻击者能够对 gadgets 进行链接.由于软件漏洞的不可避免,对代码重用攻击的攻防研究通常认为第二个条件是成立的.因而研究者们着重于如何在浏览器中寻找或注入新的 gadgets.特别注意的是,代码重用攻击所提供的任意代码执行权限将能帮助攻击者 frame 直接绕过浏览器同源策略的限制.

Snow 等[32]提出了 JIT-ROP 攻击,一种针对 JIT 代码的代码重用攻击方案.JIT-ROP 攻击假设攻击者 frame 可以通过编写合适的 JavaScript 代码来利用浏览器的漏洞,从而获取到反复读取任意内存地址的能力.攻击者可以利用该能力来跟踪程序指针.利用程序指针的指向关系,攻击者可以获取到足够多的代码内存页.接下来,攻击者可以对收集到的代码内存页进行分析,从中搜索到发起代码重用攻击所需的 gadgets 以及一些特殊的 API 函数,如加载链接库等.最后,攻击者通过利用控制流劫持漏洞来链接搜索到的 gadgets 来调用目标 API,并提供所需的参数.Snow 等在 IE 浏览器中成功发起了 JIT-ROP 攻击.

JIT-ROP 攻击的工作方式是寻找已存在代码页中的 gadagets,这并不能保证攻击者所需要的 gadgets 一定能够被找到.考虑到 JIT Spraying 攻击能够实现代码注入,Athanasakis 等[20]设计了结合 JIT Spraying 攻击与 JIT-ROP 攻击的代码重用攻击.该攻击利用 JIT Spraying 攻击中所使用的大常数来往浏览器中注入 gadgets,然后与 JIT-ROP 一样,利用控制流劫持漏洞来将注入的 gadgets 链接到一起,从而发起代码重用攻击.图 11 中展示了生成执行 mprotect 系统调用并提供相关参数所需的 gadgets 与对应的 JavaScript 代码.例如,在 JavaScript 中为变量 g3 赋值为 12828721,当 JIT 编译器对 JavaScript 进行编译后,所生成的二进制代码将包含 c3c031,该指令片段的含义是对 eax 寄存器清零后返回.

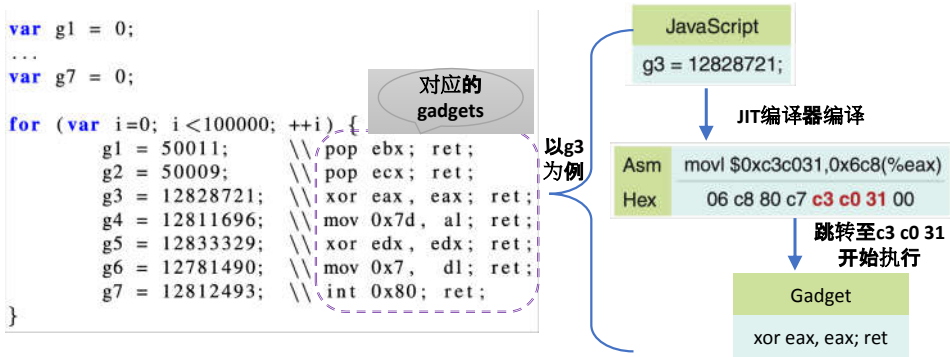
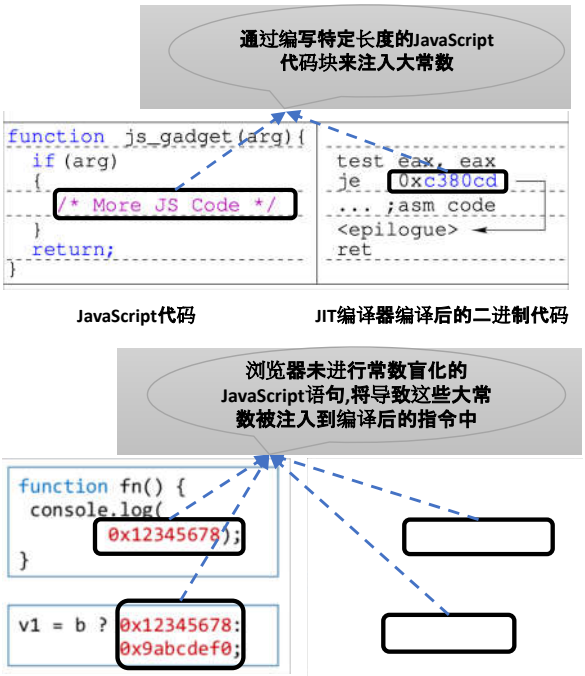


Fig 11 Code reuse attack by combining JIT Spraying and JIT-ROP attack[20]

图 11 结合 JIT Spraying 与 JIT-ROP 的代码重用攻击[20]

更进一步的,Maisuradze 等[117]发现除了 JavaScript 中的大常数外,还有很多其他方案可以实现 gadgets 的注入.举例来说,含有条件判断语句的 JavaScript 代码被 JIT 编译器编译后同样能够产生所需要的 gadgets.如图 12(a)所示,JavaScript 函数 js_gadget 中的条件语句将被编译为“test eax,eax;je 0xc380cd”.其中 JE 指令后的数字代表中 if 代码块内的指令长度,因此,通过设计相应的 if 代码块,攻击者 frame 可以在 JIT 代码中注入任意代码.Maisuradze 等[21]随后还开发了 DachShund 框架来帮助浏览器测试 JavaScript 中哪些情况会导致 gadgets 的注入.图 12(b)中展示了 DachShund 框架发现的 gadgets 注入场景,包括在 JavaScript 中调用 console 与 Math 等 API 时参数中的大常数、三元语句以及 switch 语句中的大常数等.



(a) 利用条件语句的 gadaget 注入场景[117] (b) DachShund 发现的部分 gadget 注入场景[21]

Fig 12 Examples of injecting gadgets through JavaScript

图 12 JavaScript 注入 gadgets 的方式示例

5.2.2 防御方案

应对代码重用攻击的方案有两种思路,包括对浏览器进行额外的安全加固和对 JavaScript 代码的重写.

首先,对浏览器进行修改意味着可以确保现有的网站可以在不做任何改动的情况下运行并提高安全.Athanasakis 等[20]通过对 IE 浏览器的反汇编发现 IE 中实现了常数盲化(Constant Blinding)机制来阻止大常数的注入.当 JIT 编译器发现 JavaScript 中含有大于两个字节的大常数时,将会首先生成一个随机数,然后将 JIT 代码用两条指令替代:第一条将随机数存储到指定寄存器中,第二条对寄存器作异或操作.异或操作的操作数为随机数与原来大常数的异或值.经过常数盲化后,JIT 代码中将不存在大常数所直接对应的指令,但 JIT 代码的逻辑未发生变化.特别的,考虑到两个字节的常数在 JIT 代码中大量存在与性能需求,浏览器不能对两个字节的常数进行常数盲化.Athanasakis 等[20]随后指出利用 JavaScript 函数与两字节的常数同样能注入所需的 gadgets.因此,常数盲化并不能完全抵抗代码重入攻击.

地址随机化与控制流完整性也能被应用到浏览器中来抵抗代码重入攻击.Frassetto 等[116]设计了能够抵抗

代码重入攻击的 JITGuard 框架.JITGuard 将 JIT 编译器放置在 SGX 机制所创建的 enclave 中,并且利用地址随机化与地址双向映射将 JIT 编译器编译生成的 JIT 代码放置在随机的区域内.JITGuard 确保随机区域的地址只有 enclave 内的 JIT 编译器才能获取.即使攻击者通过 JIT Spraying 攻击注入了 gadgets,攻击者仍无法定位到 gadgets 的地址,因而代码重用攻击将无法成功.Niu 等[118]将控制流完整性机制应用到 JIT 代码中,其提出的 RockJIT 机制能够抵抗代码重用攻击,但平均产生了 14.4%的运行时开销.

其次,对 frame 中 JavaScript 代码进行验证与重写也能应对代码重用攻击,并能保证浏览器本身的性能不会因为安全加固而受到影响.Maisuradze 等[21]在用户浏览器与 Web 服务器之间实现了一个网络代理,该代理对接收到的 JavaScript 代码进行重写来消除大常数.图 13 给出了消除两种形式的大常数的示例.攻击者可通过对变量的赋值来注入大常数,重写后的 JavaScript 脚本利用 JavaScript API 中的 parseInt 函数,使得大常数 0x1234 作为字符串存在内存中.注意除了明显的赋值语句外,类似于“1234”&567 的形式也会引入大常数.JIT 编译器在解析右图的 JavaScript 代码时会进行预处理,即执行“1234”&567 语句并将结果 mov 到存储变量 i 的寄存器中.网络代理可以利用 JavaScript API 中的 toString 函数要求编译器将变量作为字符串存储.为保证 JavaScript 重写的完备性,网络代理将首先解析 JavaScript 代码所对应的抽象语法树,然后遍历抽象语法树来移除大常数,最后根据更新后的抽象语法树来生成重写后的 JavaScript 代码.

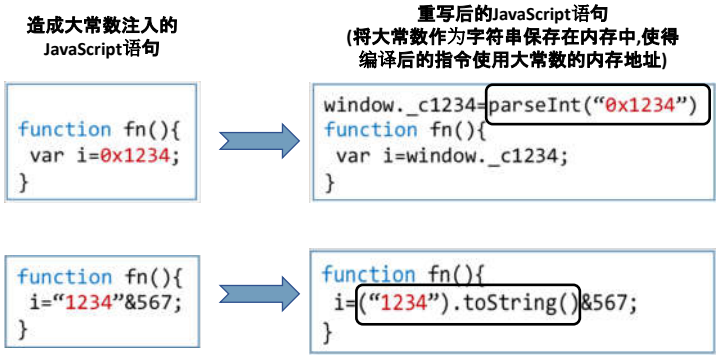


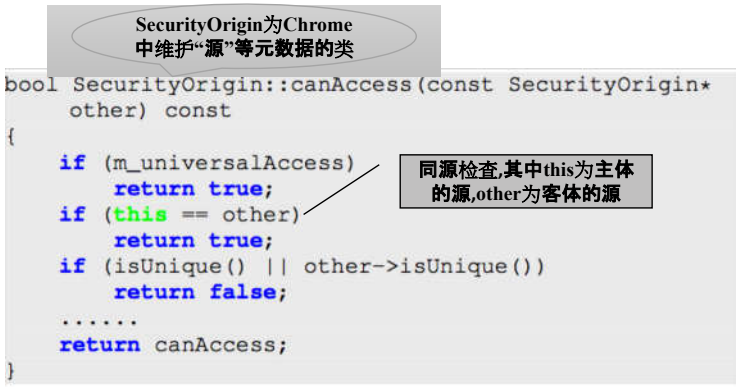
Fig 13 Examples of removing big constants (e.g., 0x1234) through rewriting JavaScript[21]

图 13 消除大常数(如 0x1234)的 JavaScript 重写机制示例[21]

5.3 仅数据攻击及防御

5.3.1 攻击方案

仅数据攻击同样能够被用来绕过或欺骗同源策略的安全监控器.Jia 等[22]发现浏览器的同源策略安全监控器依赖浏览器渲染进程维护的元数据来进行策略实施.图 14 中给出了 Chrome 浏览器的同源策略实施的部分代码,其中 SecurityOrigin 类为每个 frame 维护了对应的元数据,包括 URL 与源等信息.该代码中第二个 if 语句判断了 SecurityOrigin 对象是否相同,也即是否同源.然而,第一个 if 语句表明了如果攻击者能够通过仅数据攻击将 m_universalAccess 修改为 true,Chrome 的同源策略实施将会被绕过.Jia 等利用了 Chrome 的一个已知漏洞来使得攻击者 frame 里的 JavaScript 代码能够访问任意内存地址的数据,并且利用 Chrome 处理 HTML 标签的一些特性来绕过了浏览器中的地址随机化保护,最终使得攻击者直接定位到 m_universalAccess 的内存地址,并对其进行修改.攻击者 frame 可以内嵌 DropBox 或 Google Play 等云服务的 frame(受害者 frame).然后利用该仅数据攻击,攻击者可以绕过同源策略来在受害者 frame 中点击按钮,如从 Google Play 中下载恶意应用.此后,云服务将下载的恶意应用同步到用户本地,打破浏览器多进程模型维护的用户本地与网页的边界,实现木马程序的注入.




```
bool SecurityOrigin::canAccess(const SecurityOrigin*
     other) const
{
    if (m_universalAccess)
        return true;
    if (this == other)
        return true;
    if (isUnique() || other->isUnique())
        return false;
    ......
    return canAccess;
}
```

Fig 14 The source code of SOP enforcement in Chrome[22]

图 14 Chrome 浏览器同源策略检查部分源代码[22]

Rogowski 等[23]在 Jia 的工作的基础上,更进一步地提出了内存制图(memory cartography)技术来简化仅数据攻击的构造过程.内存制图技术包含四个步骤:首先攻击者在线下访问目标应用并执行受害者用户可能会执行的操作,与此同时,攻击者暂停浏览器运行并对渲染进程进行内存扫描来获取足够多的数据指针;此后,攻击者利用扫描到的数据指针制定内存地图,这其中需要考虑到地址的随机化问题;接下来攻击者需要找到浏览器的内存泄漏漏洞使得其可以对任意内存进行读写;最后,攻击者 frame 在运行时利用内存地图与内存泄漏漏洞来查找目标数据的地址,并对目标数据进行修改.实验表明该内存制图技术能够直接用来重现 Jia 等的工作,并在 IE 浏览器上也实现了内存利用来泄漏受害者 frame 的 cookie.

Frassetto 等[116]实现了对 JIT 编译器的仅数据攻击 DOJITA.JIT 编译器在编译 JavaScript 代码时会首先生成中间语言,然后根据中间语言来生成最后的二进制代码.如果攻击者能够在中间语言中加入一些伪造的数据,这些数据将会被编译为二进制代码执行.Frassetto 等利用浏览器的堆溢出漏洞实现了该仅数据攻击,使得攻击者 frame 获取到在渲染进程中执行任意代码的能力,同源策略也将很容易被攻击者绕过.

5.3.2 防御方案

仅数据攻击发生的根本原因是浏览器对一些安全相关敏感数据的不完全保护,使得攻击者可以利用内存漏洞来对敏感数据进行任意读写.因此,解决仅数据攻击必然需要对浏览器进行安全加固以保护安全敏感数据.

Jia 等[22]认为地址随机化能够用来保护这些安全相关的数据.浏览器可以将所有安全相关数据放置在一个单独的内存段中,并对内存段的基址进行随机化处理.更进一步的,浏览器要保证没有指针从内存段外指向内存段里,否则内存攻击者将可以跟踪指针来找到随机化后的内存段.此外,浏览器还要保证攻击者无法通过其他内存数据来引用该内存段的基址,如将内存地址放在特殊的寄存器中来避免基址的泄漏.在具体实现过程中,为避免对浏览器的大范围修改,Jia 等使用浏览器的可信内核进程来维护该内存段的基址.

Reis 等[33]对浏览器的多进程架构进行加固来保护这些安全相关的数据.传统的浏览器多进程架构会使得位于同一浏览实例的 frame 放在同一渲染进程中[24,25](见 2.1 节),所以当一个 frame 内嵌跨源子 frame 时,父子 frame 将在同一个进程中运行,同源策略的安全监控器不得不也放置在渲染进行中进行策略实施.Reis 等提出了 Site Isolation 架构,该架构彻底地将不同网站的 frame 运行在不同的渲染进程里.在这种情况下,即使攻击者 frame 对其所在的渲染进程发起了内存攻击并控制了该渲染进程,由于进程边界的存在,攻击者 frame 依旧不能访问受害者 frame 所在的渲染进程的数据,从而实现更强的安全隔离.

对安全敏感数据的保护还可以使用 SGX 等 CPU 级别的隔离机制.例如,Frassetto 等[116]等将 JS 引擎放置在利用 SGX 创建的 enclave 中,由于 enclave 外的代码无法对 enclave 里的内存进行访问,攻击者 frame 将无法修改 JavaScript 引擎中的敏感数据,之前的 DOJITA 仅数据攻击将无法实现.TrustJS[119]与 Fidelius[120]也利用硬件机制实现了类似的安全加固.

5.4 方案对比与讨论

表 8 总结了浏览器三种内存攻击的代表性工作、攻击原理、对同源策略的影响以及相关防御机制.我们从以下两个角度来分析讨论这三类内存攻击:

- 攻击原理: 对浏览器发起这三类内存攻击的共同点是利用来自于 JavaScript 引擎的漏洞,攻击者可以利用这些漏洞来影响可执行代码(如通过大常数等)、获得内存读取能力(如搜索所有代码页中的 gadgets)、以及获得内存写能力(如修改元数据).根据 Jia 等[22]对安全漏洞的分析, Chrome 于 2014 年至 2016 年间存在 599 个安全漏洞,其中 104 个漏洞允许攻击者获得内存读写能力,每个 Chrome 的版本平均存在 6 个内存漏洞.这意味着浏览器内存攻击与浏览器安全增强将是长期存在的研究内容;
- 对同源策略的影响: 对浏览器的三类内存攻击均对同源策略产生了破坏性的影响.其中,代码注入攻击与代码重用攻击的最终目的都是在渲染进程内执行任意代码,这将能够直接绕过同源策略的限制来访问跨源资源;仅数据攻击可以作用在与同源策略有关的元数据或者其他数据上,前者将导致攻击者可以欺骗同源策略的安全监控器,而后者意味着攻击者可能绕过同源策略以访问跨源资源.

Table 8 Summary on memory attacks on browser

表 8 浏览器内存攻击机制总结

攻击机制	代表性工作	攻击原理	对同源策略安全的影响	相关防御机制
代码注入攻击	Gong 等[18]; Song 等[17]	JIT 代码页的权限需在可写与可执行切换	执行任意代码,从而绕过同源策略以访问跨源资源	W^X、地址空间随机化
	Blazakis 等[19]	JIT 编译器忽略大常数		常数盲化、地址空间随机化
代码重用攻击	Snow 等[32]	利用漏洞搜索现有代码页以获取 gadgets	执行任意代码,从而绕过同源策略以访问跨源资源	地址空间随机化、控制流完整性
	Athanasakis 等[20]; Maisuradze 等[117]	利用大常数注入 gadagets		常数盲化、地址空间随机化、控制流完整性、JavaScript 重写
仅数据攻击	Frassetto 等[116]	利用漏洞修改 JIT 编译器元数据	执行任意代码,从而绕过同源策略以访问跨源资源	内存隔离、地址空间随机化
	Jia 等[22]; Rogowski 等[23]	利用漏洞修改同源策略元数据	修改元数据,欺骗同源策略以访问跨源资源	内存隔离、地址空间随机化

表 9 对抵抗以上三类攻击的防御方案进行了总结分析,包括防御机制类型、能抵抗的攻击类型、对浏览器引入的性能开销以及对浏览器代码的修改数量.我们从防御机制类型的角度来对这些方案进行总结讨论:

- W^X 与控制流完整性通常被认为是分别解决代码注入与重入攻击的有效方式.目前主流浏览器中已经实现了 W^X 保护[114].然而控制流完整性为浏览器带来了不可忽略的开销,例如 Niu 等[118]等方案引入了 14.6%的开销,这使得其在浏览器中应用还有一些距离;
- 地址空间随机化几乎能用来抵抗所有类型的内存攻击,但必须要求随机化后内存段的基址很难被攻击者找到,因此需要谨慎处理指针以及维护内存段的机制;
- 利用特殊 JIT 编译器特性的攻击可以被相应的防御机制所抵抗,如针对于大常数的常数盲化机制与 JavaScript 代码重写机制.然而,常数盲化机制引入了 15%-80%的性能开销性能的需求,这使得其不能应用于 1 到 2 字节的常数上,而使用 1 到 2 字节常数进行内存攻击是可行的[20];JavaScript 重写机制对 Chrome 与 Edge 浏览器分别引入了 21%与 24%的开销,且需要网络代理的辅助[21],这对于要求性能与易用性的浏览器来说是很难接受的;
- 内存隔离机制能够从根本上解决内存攻击问题,但通常意味着对浏览器现有架构的修改,如 Site Isoltion 架构[33]对 Chrome 修改或增加了 450k 行数的源代码,这在实现以及推广上均存在较大的问题.此外,对浏览器的架构修改也可能引入额外的开销,如何确保性能也是需要考虑的问题.

Table 9 Comparison on defenses against the memory attacks on browsers

表 9 浏览器内存防御方案对比

防御方案	防御机制类型	应对攻击	性能开销	浏览器代码修改数量
Chen 等[112]	W^X	代码注入攻击	Chrome 的 V8 引擎: 0.5%; Safari 的 JavaScript 引擎: 0.1%	3K SLOC
Crane 等[113]	W^X	代码注入攻击	Chrome 的 V8 引擎: 7.8%	-
Athanasakis 等[20]	常数盲化	代码注入攻击; 代码重用攻击	IE 的 Chakra 引擎: 15%-80%	-
Niu 等[118]	控制流完整性	代码重用攻击	Chrome 的 V8 引擎: 14.6%	811 SLOC
Maisuradze 等[21]	JavaScript 重写	代码重用攻击	Chrome: 21%; Edge: 24%	-
Frassetto 等[116]	内存隔离与地址空间随机化	代码注入攻击; 代码重用攻击; 仅数据攻击	Firefox 的 SpiderMonkey 引擎: 9.8%	2673 SLOC
Reis 等[33]	内存隔离	仅数据攻击	Chrome: 9-13% (内存开销)	450k SLOC

5.5 小结

Web 的发展与浏览器对即时编译的需求使得内存攻击者们逐渐将浏览器作为攻击目标.代码注入攻击、重用攻击与仅数据攻击的发起将会直接破坏浏览器所实施的同源策略,导致攻击者几乎可以无限制地访问受害者 frame 的资源.JavaScript 代码重写是解决内存攻击的一种方案,但性能与对目前网站的兼容性限制了其实用性.基于浏览器加固的方案将是应对内存攻击的主流,如浏览器从单进程演变到多进程模型乃至目前的 Site Iosltion 即是典型的体现.不论如何,内存攻防对抗将使得浏览器同源策略安全在曲折中增强.

6 未来研究展望

同源策略的重要性使得其被视为浏览器安全的基石,然而,Web 应用需求的多样性与浏览器本身的脆弱性使得策略攻击者与内存攻击者能够绕过或欺骗同源策略以获取跨源资源.如何解决现有安全威胁、应对未知威胁,并且兼顾易用性、灵活性、向后兼容性等多方面的因素,需要更进一步的研究,具体表现为:

(1) 易用性:应对策略攻击者的防御方案通常要求 Web 应用开发者提供细粒度或者灵活的策略,例如应对第三方脚本的防御方案[30,44,51,55].然而,Web 的多样性与复杂性使得研究者们不能简单地假设开发者都具备很强的安全意识,不能要求开发者来制定复杂的策略甚至需要考虑策略之间的一致与冲突.同源策略安全增强机制的易用性将促进其应用于实际.

(2) 灵活性:Web 环境是不断演化与进步的,新的需求与场景层出不穷,诸如用于保存历史的 Archive 网站、广告分析、用户追踪与内容安全策略等新型策略的出现使得同源策略在实际应用上捉襟见肘[10,14,31,93],未来的同源策略应当要更加灵活以支持这些新型需求与场景.

(3) 向后兼容性:同源策略作为浏览器所必须实现的基本安全策略,对其安全增强必须要考虑到向后兼容性.安全增强方案需要能够支持现有的同源策略,不能由于安全增强导致大部分网站不能渲染与运行或者必须作出对应修改.

因此,我们认为未来的研究工作应该重点关注以下几个方面:

(1) 对有潜在风险代码的权限限制

网页中潜在风险代码包括了第三方脚本与处理跨源数据的处理函数等,前者由同源策略规则不足所引起并使得第三方脚本能任意操作网页资源(见 3.1.2 节),后者由跨源通信机制所引起并使得接收方存在数据注入攻击的威胁(见 4.1.2 节).现有的研究工作的思路在于隔离第三方脚本[30,44,51,54]与推断跨源数据接收方的安全行为模型[92].这些防御机制要么为 Web 应用开发者网页编写造成不便,要么对攻击行为的检测存在误报的风险.我们认为,未来的研究工作可以对这两种情况进行统一,可以借助于 iframe sandbox 机制[121]的思路来对 JavaScript 代码的权限进行划分，进而提出统一的访问控制框架来对有潜在风险代码进行权限限制.例如,Web

开发者可以剥夺不可信第三方脚本的 DOM 元素访问权限、剥夺处理跨源数据处理函数的网络访问权限等.使用这种基于权限的控制一方面能覆盖潜在风险代码运行的整个生命周期,另一方面更便于 Web 开发者理解与使用.

(2) 同源策略与其他策略协作

同源策略是浏览器安全的基本策略,但并不是唯一策略.新型场景与需求的出现使得浏览器中出现了内容安全策略[10]等新型策略,这要求同源策略与这些新型策略之间要保持一致性,确保一种策略不能由于其他策略的存在而被绕过或者欺骗(见 3.2.1 节与 3.3.1 节).我们认为,未来的研究工作需要设计策略协作安全验证机制来辅助新型策略设计者构造策略,需要提出新的框架与系统以帮助 Web 应用开发者来为其网页配置这些策略并检查验证策略的合理性与一致性,更进一步的,未来的研究工作可能需要在浏览器中提供统一的框架来定义与实施这些策略.

(3) 数据可控的跨源资源共享

现有的跨源资源共享机制研究通常考虑共享过程中资源被窃取或者对接收方的影响等问题,而忽略共享后资源的使用权限问题.不论是服务器辅助还是无服务器辅助的机制,数据所有者在共享后将直接对数据失去控制权,数据接收方可以任意地使用、修改与传播共享的资源.例如,一个 Web 应用可能请求用户获取其它社交网站的联系人信息以进行用户推荐,而当社交网站将信息传输给该 Web 应用后,后者将可以任意地使用这些敏感信息[60].我们认为未来的跨源资源共享应当关注共享数据的可控性,如对于隐私数据可以要求数据在短暂使用后必须删除、对于共享数据的进一步传输需要经由数据所有者的同意.数据可控的跨源资源共享将极大地促进 Web 资源可靠交互.

(4) 硬件或操作系统支持的同源策略不同主体间的隔离机制

同源策略的本质上是为 Web 资源设定边界(同源),并限制跨边界的资源访问.然而,现有同源策略实施依赖于浏览器来实现主体之间的隔离.浏览器作为大型软件不可避免地面临着内存攻击,我们认为未来的研究工作应当会更倾向于使用基于操作系统甚至是基于硬件的隔离机制,以缩减可信计算基来实现更强的安全隔离.然而,可信计算基的缩减意味着策略实施将面临着一些语义的丢失,例如,Site Isolation 项目[33]利用进程来进行主体间的隔离,但是需要其他机制来判断 Web 资源类型来决定是否进行隔离.此外,这种浏览器架构上的变化还需要考虑性能以及向后兼容性等因素.

7 结束语

本文对浏览器同源策略安全的研究进展进行了深入分析和总结.首先介绍了同源策略的规则和跨域/跨源通信机制,分析了同源策略安全研究的威胁模型和研究方向;接着,分别总结分析了同源策略规则不足与应对方案、跨域与跨源通信机制安全威胁及应对方案、以及内存攻击下的同源策略安全;最后,展望了同源策略安全的未来研究方向.期望我们的工作能够为以后的研究者给予有益的参考,为同源策略安全研究作出贡献.

致谢 我们向同源策略安全研究的先行者以及对本文工作提出宝贵建议的评审老师表示衷心的感谢!

References:

[1] The Web Origin Concept. 2011. https://tools.ietf.org/html/rfc6454

[2] Same Origin Policy in W3C. 2010. https://www.w3.org/Security/wiki/Same_Origin_Policy

[3] The HTML5 spec's definition of Origin. 2020. https://html.spec.whatwg.org/multipage/origin.html#concept-origin

[4] Same Origin Policy in Wikipedia. 2020. https://en.wikipedia.org/wiki/Same-origin_policy

[5] Same Origin policy in Firefox. 2020. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[6] Nikiforakis N, Invernizzi L, Kapravelos A, et al. You are what you include: large-scale evaluation of remote javascript inclusions. In: Proc. of the 2012 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2012). 2012. 736–747. [doi: 10.1145/2382196.2382274]

[7] Zhou Y, Evans D. Understanding and Monitoring Embedded Web Scripts. In: Proc. of the 2015 IEEE Symp. on Security and Privacy (SP 2015). 2015. 850–865. [doi: 10.1109/SP.2015.57]

[8] Ikram M, Masood R, Tyson G, et al. The chain of implicit trust: An analysis of the web third-party resources loading. In: Proc. of the 2019 International Conf. on World Wide Web (WWW 2019). 2019. 2851–2857. [doi: 10.1145/3308558.3313521]

[9] Cao Y, Rastogi V, Li Z, et al. Redefining web browser principals with a configurable origin policy. In: Proc. of the 2013 Annual IEEE/IFIP International Conf. on Dependable Systems and Networks (DSN 2013). 2013. 1–12. [doi: 10.1109/DSN.2013.6575317]

[10] Somé D F, Bielova N, Rezk T. On the content security policy violations due to the same-origin policy. In: Proc. of the 2017 International Conf. on World Wide Web (WWW 2017). 2017. 877–886. [doi: 10.1145/3038912.3052634]

[11] Barth A, Jackson C, Mitchell J C. Robust defenses for cross-site request forgery. In: Proc. of the 2008 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2008). 2008. 75–88. [doi: 10.1145/1455770.1455782]

[12] Lekies S, Stock B, Wentzel M, et al. The Unexpected Dangers of Dynamic JavaScript. In: Proc. of the 2015 USENIX Security Symp. (USENIX Security 2015). 2015. 723–735.

[13] Singh K, Moshchuk A, Wang H J, et al. On the Incoherencies in Web Browser Access Control Policies. In: Proc. of the 2010 IEEE Symp. on Security and Privacy (SP 2010). 2010. 463–478. [doi: 10.1109/SP.2010.35]

[14] Son S, Shmatikov V. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In: Proc. of the 2013 Network and Distributed System Security Symp. (NDSS 2013). 2013.

[15] Popescu P. Practical JSONP Injection. 2017. https://securitycafe.ro/2017/01/18/practical-jsonp-injection/

[16] Chen J, Jiang J, Duan H, et al. We Still Don't Have Secure Cross-Domain Requests: an Empirical Study of CORS. In: Proc. of the 2018 USENIX Security Symp. (USENIX Security 2018). 2018. 1079–1093.

[17] Song C, Zhang C, Wang T, et al. Exploiting and Protecting Dynamic Code Generation. In: Proc. of the 2015 Network and Distributed System Security Symp. (NDSS 2015). 2015.

[18] Gong G. Pwn a Nexus Device With a Single Vulnerability. 2016. https://cansecwest.com/slides/2016/CSW2016_Gong_Pwn_a_Nexus_device_with_a_single_vulnerability.pdf

[19] Blazakis D. Interpreter exploitation. In: Proc. of the 2010 USENIX Conf. on Offensive technologies. (WOOT 2010). 2010. 1–9.

[20] Athanasakis M, Athanasopoulos E, Polychronakis M, et al. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In: Proc. of the 2015 Network and Distributed System Security Symp. (NDSS 2015). 2015.

[21] Maisuradze G, Backes M, Rossow C. Dachshund: Digging for and Securing (Non-) Blinded Constants in JIT Code. In: Proc. of the 2017 Network and Distributed System Security Symp. (NDSS 2017). 2017.

[22] Jia Y, Chua Z L, Hu H, et al. The Web/Local Boundary Is Fuzzy: A Security Study of Chrome's Process-based Sandboxing. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2016). 2016. 791–804. [doi: 10.1145/2976749.2978414]

[23] Rogowski R, Morton M, Li F, et al. Revisiting browser security in the modern era: New data-only attacks and defenses. In: Proc. of the 2017 IEEE European Symp. on Security and Privacy (EuroS&P 2017). 2017. 366–381. [doi: 10.1109/EuroSP.2017.39]

[24] Reis C, Gribble S D. Isolating web programs in modern browser architectures. In: Proc. of the 2009 ACM European Conf. on Computer systems. (EuroSys 2009). 2009. 219–232. [doi: 10.1145/1519065.1519090]

[25] Chromium's Process Models. https://www.chromium.org/developers/design-documents/process-models

[26] Cross-site Scripting (XSS). https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[27] Lekies S, Stock B, Johns M. 25 million flows later: large-scale detection of DOM-based XSS. In: Proc. of the 2013 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2013). 2013. 1193–1204. [doi: 10.1145/2508859.2516703]

[28] Stock B, Lekies S, Mueller T, et al. Precise Client-side Protection against DOM-based Cross-Site Scripting. In: Proc. of the 2014 USENIX Security Symp. (USENIX Security 2014). 2014. 655–670.

[29] Melicher W, Das A, Sharif M, et al. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. In: Proc. of the 2018 Network and Distributed System Security Symp. (NDSS 2018). 2018.

[30] Ingram L, Walfish M. TreeHouse: JavaScript sandboxes to helpWeb developers help themselves. In: Proc. of the 2012 USENIX Conf. on Annual Technical Conference. (ATC 2012). 2012.

[31] Lerner A, Kohno T, Roesner F. Rewriting history: Changing the archived web from the present. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2017). 2017. 1741–1755. [doi: 10.1145/3133956.3134042]

[32] Snow K Z, Monrose F, Davi L, et al. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: Proc. of the 2013 IEEE Symp. on Security and Privacy (SP 2013). 2013. 574–588. [doi: 10.1109/SP.2013.45]

[33] Reis C, Moshchuk A, Oskov N. Site isolation: process separation for web sites within the browser. In: Proc. of the 2019 USENIX Security Symp. (USENIX Security 2019). 2019. 1661–1678.

[34] Yue C, Wang H. Characterizing insecure javascript practices on the web. In: Proc. of the 2009 International Conf. on World Wide Web (WWW 2009). 2009. 961–970. [doi: 10.1145/1526709.1526838]

[35] Stock B, Pfistner S, Kaiser B, et al. From facepalm to brain bender: Exploring client-side cross-site scripting. In: Proc. of the 2015 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2015). 2015. 1419–1430. [doi: 10.1145/2810103.2813625]

[36] Lauinger T, Chaabane A, Arshad S, et al. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In: Proc. of the 2017 Network and Distributed System Security Symp. (NDSS 2017). 2017.

[37] Kumar D, Ma Z, Durumeric Z, et al. Security challenges in an increasingly tangled web. In: Proc. of the 2017 International Conf. on World Wide Web (WWW 2017). 2017. 677–684. [doi: 10.1145/3038912.3052686]

[38] Laperdrix P, Rudametkin W, Baudry B. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In: Proc. of the 2016 IEEE Symp. on Security and Privacy (SP 2016). 2016. 878–894. [doi: 10.1109/SP.2016.57]

[39] Hayes J, Danezis G. k-fingerprinting: A robust scalable website fingerprinting technique. In: Proc. of the 2016 USENIX Security Symp. (USENIX Security 2016). 2016. 1187–1203.

[40] Wang T, Goldberg I. Walkie-talkie: An efficient defense against passive website fingerprinting attacks. In: Proc. of the 2017 USENIX Security Symp. (USENIX Security 2017). 2017. 1375–1390.

[41] Vastel A, Laperdrix P, Rudametkin W, et al. FP-STALKER: Tracking browser fingerprint evolutions. In: Proc. of the 2018 IEEE Symp. on Security and Privacy (SP 2018). 2018. 728–741. [doi: 10.1109/SP.2018.00008]

[42] Vastel A, Laperdrix P, Rudametkin W, et al. FP-scanner: the privacy implications of browser fingerprint inconsistencies. In: Proc. of the 2018 USENIX Security Symp. (USENIX Security 2018). 2018. 135–150.

[43] Patra J, Dixit P N, Pradel M. Conflictjs: finding and understanding conflicts between javascript libraries. In: Proc. of the 2018 International Conf. on Software Engineering. (ICSE 2018). 2018. 741–751. [doi: 10.1145/3180155.3180184]

[44] Tran T, Pelizzi R, Sekar R. Jate: Transparent and efficient javascript confinement. In: Proc. of the 2015 Annual Computer Security Applications Conf. (ACSAC 2015). 2015. 151–160. [doi: 10.1145/2818000.2818019]

[45] Agten P, Van Acker S, Brondsema Y, et al. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In: Proc. of the 2012 Annual Computer Security Applications Conf. (ACSAC 2012). 2012. 1–10. [doi: 10.1145/2420950.2420952]

[46] Musch M, Steffens M, Roth S, et al. ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices. In: Proc. of the 2019 ACM Asia Conf. on Computer and Communications Security (Asia CCS 2019). 2019. 391–402. [doi: 10.1145/3321705.3329841]

[47] Reis C, Dunagan J, Wang H J, et al. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In: ACM Transactions on the Web (TWEB 2007). 2007. [doi: 10.1145/1281480.1281481]

[48] Miller M S, Samuel M, Laurie B, et al. Safe active content in sanitized JavaScript. In: Google Inc. Tech. Report. 2008.

[49] Guarnieri S, Livshits V B. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In: Proc. of the 2009 USENIX Security Symp. (USENIX Security 2009). 2009. 78–85.

[50] Ter Louw M, GaneshK T, VenkatakrishnanV N. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In: Proc. of the 2010 USENIX Security Symp. (USENIX Security 2010). 2010. 371–388.

[51] Mickens J. Pivot: Fast, synchronous mashup isolation using generator chains. In: Proc. of the 2014 IEEE Symp. on Security and Privacy (SP 2014). 2014. 261–275. [doi: 10.1109/SP.2014.24]

[52] Luo Z, Rezk T. Mashic compiler: Mashup sandboxing based on inter-frame communication. In: Proc. of the 2012 IEEE Computer Security Foundations Symp. (CSF 2012). 2012. 157–170. [doi: 10.1109/CSF.2012.22]

[53] Wang H J, Fan X, Howell J, et al. Protection and communication abstractions for web browsers in MashupOS. In: ACM SIGOPS Operating Systems Review. 2007. 1–16. [doi: 10.1145/1323293.1294263]

[54] Dong X, Tran M, Liang Z, et al. AdSentry: comprehensive and flexible confinement of JavaScript-based advertisements. In: Proc. of the 2011 Annual Computer Security Applications Conf. (ACSAC 2011). 2011. 297–306. [doi: 10.1145/2076732.2076774]

[55] Meyerovich L A, Livshits B. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In: Proc. of the 2010 IEEE Symp. on Security and Privacy (SP 2010). 2010. 481–496. [doi: 10.1109/SP.2010.36]

[56] Van Acker S, De Ryck P, Desmet L, et al. WebJail: least-privilege integration of third-party components in web mashups. In: Proc. of the 2011 Annual Computer Security Applications Conf. (ACSAC 2011). 2011. 307–316. [doi: 10.1145/2076732.2076775]

[57] Window.postMessage JavaScript API. 2019. https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage

[58] Moshchuk A, Wang H J, Liu Y. Content-based isolation: rethinking isolation policy design on client systems. In: Proc. of the 2013 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2013). 2013. 1167–1180. [doi: 10.1145/2508859.2516722]

[59] Barth A, Jackson C, Mitchell J C. Securing Frame Communication in Browsers. In: Proc. of the 2008 USENIX Security Symp. (USENIX Security 2008). 2008.

[60] Stefan D, Yang E Z, Marchenko P, et al. Protecting Users by Confining JavaScript with COWL. In: Proc. of the 2014 USENIX Symp. on Operating Systems Design and Implementation (OSDI 2014). 2014. 131–146.

[61] Stefan D, Alkhelaifi A. W3C draft for COWL. 2017. https://w3c.github.io/webappsec-cowl/

[62] Jayaraman K, Du W, Rajagopalan B, et al. Escudo: A fine-grained protection model for web browsers. In: Proc. of the 2010 IEEE International Conf. on Distributed Computing Systems. (ICDCS 2010). 2010. 231–240. [doi: 10.1109/ICDCS.2010.71]

[63] Luo T, Du W. Contego: Capability-based access control for web browsers. In: Proc. of the 2011 International Conference on Trust and Trustworthy Computing. (Trust 2011). 2011. 231–238. [doi: 10.1007/978-3-642-21599-5_17]

[64] Steven S I, Bellovin S M. Building a Secure Web Browser. In: Proc. of the 2001 USENIX Conf. on Annual Technical Conference, FREENIX Track. (ATC 2001). 2001.

[65] Cox R S, Hansen J G, Gribble S D, et al. A safety-oriented platform for web applications. In: Proc. of the 2006 IEEE Symp. on Security and Privacy (SP 2006). 2006. 350–364. [doi: 10.1109/SP.2006.4]

[66] Karlof C, Shankar U, Tygar J D, et al. Dynamic pharming attacks and locked same-origin policies for web browsers. In: Proc. of the 2007 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2007). 2007. 58–71. [doi: 10.1145/1315245.1315254]

[67] Dong X, Chen Z, Siadati H, et al. Protecting sensitive web content from client-side vulnerabilities with CRYPTONS. In: Proc. of the 2013 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2013). 2013. 1311–1324. [doi: 10.1145/2508859.2516743]

[68] Pan X, Cao Y, Liu S, et al. Cspautogen: Black-box enforcement of content security policy upon real-world websites. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2016). 2016. 653–665. [doi: 10.1145/2976749.2978384]

[69] Luo M, Laperdrix P, Honarmand N, et al. Time Does Not Heal All Wounds: A Longitudinal Analysis of Security-Mechanism Support in Mobile Browsers. In: Proc. of the 2019 Network and Distributed System Security Symp. (NDSS 2019). 2019.

[70] Calzavara S, Rabitti A, Bugliesi M. Semantics-based analysis of content security policy deployment. In: ACM Transactions on the Web (TWEB 2018). 2018. 1–36. [doi: 10.1145/3149408]

[71] Roth S, Barron T, Calzavara S, et al. Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies. In: Proc. of the 2020 Network and Distributed System Security Symp. (NDSS 2020). 2020.

[72] Calzavara S, Rabitti A, Bugliesi M. CCSP: Controlled Relaxation of Content Security Policies by Runtime Policy Composition. In: Proc. of the 2017 USENIX Security Symp. (USENIX Security 2017). 2017. 695–712.

[73] Weichselbaum L, Spagnuolo M, Lekies S, et al. CSP is dead, long live CSP! On the insecurity of whitelists and the future of content security policy. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2016). 2016. 1376–1387. [doi: 10.1145/2976749.2978363]

[74] Sudhodanan A, Carbone R, Compagna L, et al. Large-Scale Analysis & Detection of Authentication Cross-Site Request Forgeries. In: Proc. of the 2017 IEEE European Symp. on Security and Privacy (EuroS&P 2017). 2017. 350–365. [doi: 10.1109/EuroSP.2017.45]

[75] Grossman J. Advanced Web Attack Techniques using GMail. 2006. http://lists.webappsec.org/pipermail/websecurity_lists.webappsec.org/2006-January/000772.html

[76] Kern C, Kesavan A, Daswani N. Foundations of security: what every programmer needs to know. 2007.

[77] Terada T. Identifier based XSSI attacks.2015. https://www.mbsd.jp/Whitepaper/xssi.pdf

[78] Zalewski M. The tangled Web: A guide to securing modern web applications. 2012.

[79] Staicu C-A, Pradel M. Leaky images: targeted privacy attacks in the web. In: Proc. of the 2019 USENIX Security Symp. (USENIX Security 2019). 2019. 923–939.

[80] Harold E R. Block Referer headers in Firefox. 2006. http://cafe.elharo.com/privacy/privacy-tip-3-block-referer-headers-in-firefox/

[81] Johnson A. The Referer header, intranets and privacy. 2007. http://cephas.net/blog/2007/02/06/the-referer-header-intranets-and-privacy/

[82] Franken G, Goethem T Van, Joosen W. Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-Party Cookie Policies. 2018. Proc. of the 2018 USENIX Security Symp. (USENIX Security 2018). 2018. 151–168.

[83] Franken G, Van Goethem T, Joosen W. Exposing Cookie Policy Flaws Through an Extensive Evaluation of Browsers and Their Extensions. In: Proc. of the 2019 IEEE Symp. on Security and Privacy (SP 2019). 2019. 25–34. [doi: 10.1109/MSEC.2019.2909710]

[84] Pellegrino G, Johns M, Koch S, et al. Deemon: Detecting CSRF with dynamic analysis and property graphs. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2017). 2017. 1757–1771. [doi: 10.1145/3133956.3133959]

[85] Calzavara S, Conti M, Focardi R, et al. Mitch: A machine learning approach to the black-box detection of CSRF vulnerabilities. In: Proc. of the 2019 IEEE European Symp. on Security and Privacy (EuroS&P 2019). 2019. 528–543. [doi: 10.1109/EuroSP.2019.00045]

[86] Calzavara S, Conti M, Focardi R, et al. Machine Learning for Web Vulnerability Detection: The Case of Cross-Site Request Forgery. In: Proc. of the 2020 IEEE Symp. on Security and Privacy (SP 2020). 2020.

[87] document.domain JavaScript API. 2020. https://developer.mozilla.org/en-US/docs/Web/API/Document/domain

[88] Cross-Origin Resource Sharing (CORS). 2020. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[89] Ippolito B. Remote JSON – JSONP. 2005. https://bob.ippoli.to/archives/2005/12/05/remote-json-jsonp/

[90] Stock B, Johns M, Steffens M, et al. How the Web tangled itself: Uncovering the history of client-side Web (in) security. In: Proc. of the 2017 USENIX Security Symp. (USENIX Security 2017). 2017. 971–987.

[91] Cross-document messaging. 2019. https://html.spec.whatwg.org/multipage/web-messaging.html#crossDocumentMessages

[92] Weissbacher M, Robertson W K, Kirda E, et al. ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities. In: Proc. of the 2015 USENIX Security Symp. (USENIX Security 2015). 2015. 737–752.

[93] Guan C, Sun K, Wang Z, et al. Privacy breach by exploiting postmessage in html5: Identification, evaluation, and countermeasure. In: Proc. of the 2016 ACM Asia Conf. on Computer and Communications Security (Asia CCS 2016). 2016. 629–640. [doi: 10.1145/2897845.2897901]

[94] Guan C, Sun K, Lei L, et al. DangerNeighbor attack: Information leakage via postMessage mechanism in HTML5. Computers & Security. 2019. 291–305. [doi: 10.1016/j.cose.2018.09.010]

[95] Fetch Standard. 2020. https://fetch.spec.whatwg.org/

[96] Jackson C, Wang H J. Subspace: secure cross-domain communication for web mashups. In: Proc. of the 2007 International Conf. on World Wide Web (WWW 2007). 2007. 611–620. [doi: 10.1145/1242572.1242655]

[97] Liu J, Su PR, Yang M, et al. Software and Cyber Security-A Survey. Journal of Software. 2018. 29(1):42-68 (in Chinese with English abstract). [doi: 10.13328/j.cnki.jos.005320]

[98] One A. Smashing the stack for fun and profit. 1996. https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

[99] Microsoft's Data Execution Prevention. 2004. https://ohdcs.hospitality.oracleindustry.com/OperaHelp/microsoft_s_data_extraction_prevention.htm

[100] Shacham H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proc. of the 2007 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2007). 2007. 552–561. [doi: 10.1145/1315245.1315313]

[101] Roemer R, Buchanan E, Shacham H, et al. Return-Oriented Programming: Systems, Languages, and Applications. In: ACM Transactions on Information and System Security (TISSEC 2012). 2012. 1–34. [doi: 10.1145/2133375.2133377]

[102] Bletsch T, Jiang X, Freeh V W, et al. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In: Proc. of the 2011 ACM Asia Conf. on Computer and Communications Security (Asia CCS 2011). 2011. 30–40. [doi: 10.1145/1966913.1966919]

[103] Checkoway S, Davi L, Dmitrienko A, et al. Return-oriented programming without returns. In: Proc. of the 2010 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2010). 2010. 559–572. [doi: 10.1145/1866307.1866370]

[104] Abadi M, Budiu M, Erlingsson Ú, et al. Control-flow integrity. In: Proc. of the 2005 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2005). 2005. [doi: 10.1145/1102120.1102165]

[105] Abadi M, Budiu M, Erlingsson Ú, et al. Control-flow integrity principles, implementations, and applications. In: ACM Transactions on Information and System Security (TISSEC 2009). 2009. 1–40. [doi: 10.1145/1609956.1609960]

[106] Pappas V, Polychronakis M, Keromytis A D. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In: Proc. of the 2013 USENIX Security Symp. (USENIX Security 2013). 2013. 447–462.

[107] Cheng Y, Zhou Z, Miao Y, et al. ROPecker: A generic and practical approach for defending against ROP attack. In: Proc. of the 2014 Network and Distributed System Security Symp. (NDSS 2014). 2014.

[108] Onarlioglu K, Bilge L, Lanzi A, et al. G-Free: defeating return-oriented programming through gadget-less binaries. In: Proc. of the 2010 Annual Computer Security Applications Conf. (ACSAC 2010). 2010. 49–58. [doi: 10.1145/1920261.1920269]

[109] Bhatkar S, DuVarney D C, Sekar R. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In: Proc. of the 2003 USENIX Security Symp. (USENIX Security 2003). 2003. 291–301.

[110] Hu H, Shinde S, Adrian S, et al. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In: Proc. of the 2016 IEEE Symp. on Security and Privacy (SP 2016). 2016. 969–986. [doi: 10.1109/SP.2016.62]

[111] Castro M, Costa M, Harris T. Securing software by enforcing data-flow integrity. In: Proc. of the 2006 USENIX Symp. on Operating Systems Design and Implementation (OSDI 2006). 2006. 147–160.

[112] Chen P, Fang Y, Mao B, et al. JITDefender: A defense against JIT spraying attacks. In: Future Challenges in Security and Privacy for Academia and Industry (SEC 2011). 2011. 142–153. [doi: 10.1007/978-3-642-21424-0_12]

[113] Crane S, Liebchen C, Homescu A, et al. Readactor: Practical code randomization resilient to memory disclosure. In: Proc. of the 2015 IEEE Symp. on Security and Privacy (SP 2015). 2015. 763–780. [doi: 10.1109/SP.2015.52]

[114] W^X JIT-code enabled in Firefox. 2015. https://jandemooij.nl/blog/2015/12/29/wx-jit-code-enabled-in-firefox/

[115] Chen P, Wu R, Mao B. JITSafe: a framework against Just-in-time spraying attacks. In: IET Information Security. 2013. 283–292. [doi: 10.1049/iet-ifs.2012.0142]

[116] Frassetto T, Gens D, Liebchen C, et al. JITGuard: Hardening Just-in-time Compilers with SGX. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2017). 2017. 2405–2419. [doi: 10.1145/3133956.3134037]

[117] Maisuradze G, Backes M, Rossow C. What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In: Proc. of the 2016 USENIX Security Symp. (USENIX Security 2016). 2016. 139–156.

[118] Niu B, Tan G. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In: Proc. of the 2014 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2014). 2014. 1317–1328. [doi: 10.1145/2660267.2660281]

[119] Goltzsche D, Wulf C, Muthukumaran D, et al. Trustjs: Trusted client-side execution of javascript. In: Proc. of the 2017 European Workshop on Systems Security. (EuroSec 2017). 2017. 1–6. [doi: 10.1145/3065913.3065917]

[120] Eskandarian S, Cogan J, Birnbaum S, et al. Fidelius: Protecting user secrets from compromised browsers. In: Proc. of the 2019 IEEE Symp. on Security and Privacy (SP 2019). 2019. 264–280. [doi: 10.1109/SP.2019.00036]

[121] West M. Play safely in sandboxed IFrames. 2013. https://www.html5rocks.com/en/tutorials/security/sandboxed-iframes/

附中文参考文献:

[97] 刘剑, 苏璞睿, 杨珉等. 软件与网络安全研究综述. 软件学报. 2018, 29(1): 42–68.