Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matej Vehar

Varovanje kode na odjemalcu z analizo in praktično uporabo principov CIA

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: prof. dr. Denis Trček

Ljubljana, 2017

Zahvala

Zahvaljujem se moji družini in punci, ki so mi stali ob strani in me spodbujali k zaključitvi magistrskega dela. Zahvaljujem se tudi mentorju prof. dr. Denis Trčku za nasvete in pomoč pri pripravi dela. Zahvala gre tudi abbito.si in Vesni Kovjanić Janković, uni. dipl. hrvatistki, srbistki, makedonistki in rusistki, inž. medijske produkcije za lektoriranje in slovnične popravke.

Matej Vehar, 2017

"As I review the events of my past life I realize how subtle are the influences that shape our destinies."

— Nikola Tesla

Contents

Seznam uporabljenih kratic

kratica	angleško	slovensko
AJAX	Asynchronus JavaScript and XML	asinhroni JavaScript in XML
CIA	Confidentiality, Integirity and Availability	zaupnost, celovitost, razpoložljivost
CORS	Cross Origin Resource Sharing	deljenje virov med izvori
CSS	Cascading Style Sheets	oblikovna določila
DOM	Document Object Model	dokumentno objektni model
HTML	Hypertext Markup Language	označevalni jezik za oblikovanje večpredstavnostnih dokumentov
RTC	Real Time Communication	komunikacija v realnem času
SRI	SubResource Integrity	celovitost pod virov
URI	Uniform Resource Identifier	enotni identifikator vira
URL	Uniform Resource Locator	enolični naslov vira
XSS	Cross-Site Scripting	lokacijsko prestopno skriptiranje
XML	Extensible Markup Language	razširljivi označevalni jezik
CSRF	Cross-Site Request Forgery	ponarejanje spletnih zahtev
OWASP	The Open Web Application Security Project	Odprt spletno aplikacijski varnostni projekt
POST	POST HTTP Request	HTTP zahtevek tipa "POST"
GET	GET HTTP Request	HTTP zahtevek tipa "GET"
HTTPS	HyperText Transport Protocol Secure sockets	varovan komunikacijskih protokol za prenos hipertexta
HTTP	HyperText Transport Protocol	protokol za izmenjavo hiperteksta
CSP	Content Security Policy	vsebinska varnostna politika
MIME	Multipurpose Internet Mail Extension	večnamenska razširitev za elektronsko pošto
CDN	Content Delivery Network	omrežje za dostavo vsebin
FBML	Facebook Markup Language	Facebook različica hipertexta
FBJS	Facebook JavaScript	Facebook različica jezika JavaScript
SES	Secure ECMAScript Subset	varna podmnožica specifikacije ECMAScript
NVME	Non-Volatile Memory Express	hitri trajni pomnilnik

Povzetek

Naslov: Varovanje kode na odjemalcu z analizo in praktično uporabo principov CIA

Splet je postal nepredstavljiv brez tehnologij, kot je JavaScript [1]. Več kot 94% spletnih strani [2] vsebuje dinamično vsebino, ki ima vedno več zmožnosti na odjemalcih [3, 4]. Spletne strani so postale interaktivna zmes, ki vsebuje zunanje knjižnice, gradnike z oglasi in uporabniško vsebino [5, 6] ter so prikazane v brskalnikih, ki imajo prav tako nameščene dodatke iz zunanjih virov [7]. Vsi ti zunanji viri lahko predstavljajo vstopno točko za poljubno potencialno zlonamerno vsebino, ki lahko spremeni delovanje spletne strani ali jo zlorabi podatke na njej [8].

V magistrskem delu bodo raziskani obstoječi pristopi za doseganje večje varnosti pri razvoju spletne vsebine oziroma programske kode za odjemalce. Na podlagi ugotovitev iz obstoječih ukrepov in pregleda pogostih funkcionalnosti, ki jih zlorabljajo XSS napadi bo pripravljena knjižnica za okrepitev zaupnosti, integritete in razpoložljivosti (t.i. CIA triada) občutljivih funkcij okolja spletne aplikacije. Knjižnica bo temeljila na objektno zmožnostnem modelu [9] in bo nadaljevanje idej nekaterih obstoječih rešitev [10, 11].

Knjižnica bo prilagodila okolje v katerem se izvaja vsebina, zato bodo izvedeni testi, ki bodo preverili vpliv na delovanje posameznih zaščitenih funkcij.

Ključne besede

informacijska varnost, odjemalec, JavaScript, zmožnostni model

Abstract

Title: Client-side code security analysis and practical application of CIA principles

World Wide Web has become unimaginable without technologies such as JavaScript. More than 94% of web sites [2] use dynamic content, which has increasingly powerful capabilities on clients [3, 4]. Web pages have become mashup of third party libraries, widgets with ads and user generated content [5, 6] that executes in browsers with enabled third party extensions [7]. All those external dependencies are potential entry point for unwanted and malicious code, which can alter page functionalities or abuse sensitive content [8].

In this master thesis we will analyse existing functionalities and approaches to increase security of web pages. Based on the outcomes and the overview of the most abused functions by the XSS attacks we will construct a program library. Purpose of the library will be to enhance security, integrity and availability of the sensitive functions within execution environment of the web content. The Design will be based on the object-capabilities model [9] and will manifest proposed ideas by similar approaches [10, 11].

Since the library will modify execution environment, all modified functions will be tested for execution overhead.

Keywords

information security, client, JavaScript, capability model

Poglavje 1

Uvod

Splet je dan danes postal nepredstavljiv brez tehnologij, kot je JavaScript [1]. Študije [2] so pokazale, da preko 94% spletnih strani vsebuje dinamično vsebino za ali izboljšanje uporabniške izkušnje in iteraktivnosti ali pa za dostop do nabora funkcionalnosti, ki jih brskalniki ponujajo. Sodobne spletne strani lahko komunicirajo z ostalimi napravami, imajo dostop do odjemalčevih podsistemov ter nanje shranjujejo podatke [3, 4].

Drug trend v spletnih tehnologijah je sestava zmesi vsebin iz različnih virov. To so lahko zunanje knjižnice (angl. third party libraries), uporabniške vsebine na forumih, integrirane spletne vsebine, kot na primer oglasi [6, 5]. Vsi ti zunanji viri lahko predstavljajo vstopno točko za poljubno vsebino, da se izvede v uporabniški seji. Dodatno brskalniki podpirajo vstavljanje spletne vsebine s strani razširitev [7]. Vstavljena vsebina ima dostop do vmesnikov naprave, občutljivih podatkov spletnega portala in uporabnika ter lahko izvede napade v omrežju [8].

V magistrskem delu se bom poglobil v obstoječe tehnike za izboljšanje vidikov CIA [12] primarne spletne vsebine in okolja, v katerem se izvaja. Primarna vsebina je vsa spletna vsebina, ki jo razvijalec namerno vključi in deluje v skladu z lastnim opisom delovanja. Ostala vsebina - agenti - je posledica vstavitev vsebin iz nepredvidenih virov, kot so na primer razširitev brskalnika, zlonamerne vsebine iz napadov XSS [8] ali celo včasih s strani

samega uporabnika preko razvojnih orodij brskalnika.

V prvem delu bo predstavljen model brskalnika in model groženj. Nato bodo predstavljeni obstoječi pristopi in knjižnice, ki jih lahko razvijalec uporabi pri zasnovi obogatenih spletnih aplikacij (angl. rich web applications), ki vsebujejo vključitev tuje vsebine in obenem gostujejo občutljivo vsebino. Ti pristopi vključujejo uporabo internih konstruktov (okvirjev) za izolacijo tuje vsebine, politike za omejitev zunanjih komunikacij, tehnik za zagotavljanje integritete virov, filtriranje vsebine ter dinamični pristop na osnovi objektno zmožnostnega modela jezika JavaScript.

Ker se spletne tehnologije spreminjajo in razvijajo, bom preveril, ali opisani pristopi zagotavljajo ustrezen nivo kontrole vidikov CIA ter ali jih je mogoče posodobiti. Najbolj dinamičen in celosten pristop predstavlja objektno zmožnostni model, vendar so predstavljene tehnike ali samo teoretične, zahtevne za implementacijo ali pa osnovane na zastarelih konstruktih.

V osrednjem delu bom predstavil zasnovo objektno zmožnostnega modela in naravo pogona in jezika JavaScript, ki omogoča implementacijo kontrol opisanega modela. Na podlagi napadov XSS sem identificiral občutljive funkcionalnosti in vmesnike v najpogostejših brskalnikih, do katerih ima vsebina dostop. Obenem sem dodal morebitne dodatne kontrole, ki bi jih lahko razvojnik vzpostavil za izboljšanje varovanja in celovitosti vsebine.

Za zaščito občutljivih konstruktov bodo zasnovani mehanizmi zaščite in politike, ki omogočajo prilagoditev kontrol. Mehanizmi skupaj s politikami predstavljajo dele knjižnice, odpravljajo nezaželene akcije oziroma dovoljujejo kontrolirano uporabo. Mehanizmi bodo zasnovani tako, da ne bodo potrebne nikakršne prilagoditve same vsebine, edino razvojnik mora zagotoviti ustrezno uporabo knjižnice in politik.

Knjižnica deluje z nadomestitvijo lastnosti in metod z vmesnimi funkcijami, zato bodo izvedeni testi, ki bodo prikazali vpliv mehanizmov in politik na delovanje spletne vsebine. Pripravljena koda in testi so objavljeni na spletnem mestu GitHub
(`https://github.com/mvehar/JavaScriptSecurityEnchantments`).

1.1 Pregled sorodnih del

Problem omejevanja izvajanja škodljive kode že dolgo obstaja v vseh panogah računalništva. Rast spletnih tehnologij je temeljila na hitrem razvoju novih funkcionalnosti, ki so olajševale razvoj ali pa izboljšale uporabniško izkušnjo [13]. Po pojavitvi prvih napadov so bili popravki bolj odgovor na posamezne težave kot celostne rešitve. Protivirusni programi in požarni zidovi so bili med prvimi pristopi, ki so sumljivo programsko kodo odstranjevali, preden je le ta prispela do odjemalca [14].

Vendar je v določenih primerih bilo potrebno vključevati neznane vsebine v spletni portal (npr. oglasi, forumi, uporabniške strani) in takrat so se razvojniki lahko zatekli kvečjemu k uporabi okvirjev ter filtriranju vstavljene vsebine [15, 16]. Zaradi dinamične narave jezika JavaScript in dinamike pri zapisovanju programske kode v različnih formatih so se pojavili napadi, ki so te težave obšli [17, 14].

Zato so raziskovalci iskali nove načine analiziranja in omejevanja programske kode. Izvajali so poskuse s statičnimi analizami in barvanjem programskih tokov [18]. Za uporabo teh metod je bilo potrebno veliko predpriprav, kjer je bilo treba določiti, kateri deli kode so nezaželeni ter ali koda nenamerno spreminja obstoječe funkcionalnosti [19]. Poskusi so bili opravljeni tudi z vstavljanje sprotnih preverjanj izvajanja kode, vendar, ker je vsebina znotraj brskalniškega okna dinamična in enakovredna med sabo, je bilo ta preverjanja mogoče enostavno odstraniti [20].

Rešitve s strani brskalnikov so poleg okvirov (ang. frames) bile predvsem uvedbe vsebinskih varnostnih politik - določanje pravic, ki jih ima posamezni izvor in tip vsebine v brskalniku [21, 22]. Obstajajo pa tudi rešitve, ki uporabljajo zmožnosti navideznega stroja JavaScript tako, da svojo implementacijo izvedejo znotraj zaprtih funkcij (ang. closures) ali vstavijo varnejšo implementacijo metod, ki lahko zagotavljajo ustrezno delovanje [11, 10, 23, 24].

JavaScript je v tem času dobil nove funkcionalnosti, ki omogočajo zaklepanje objektov in njihovih parametrov (ang. immutable objects) [3]. Zunanjim virom kode lahko dodamo tudi kriptografski izvleček, ki zagotavlja, da

vključen vir ni bil spremenjen [25]. Obenem so se na straneh, ki prikazujejo neznano vsebino, pojavile rešitve, ki izvedejo prečiščevanje vsebine pred vstavitvijo [26, 10] ter odstranijo uporabo težavnih funkcij ali pa uporabljajo vedno nove funkcionalnosti brskalnikov za zaščito vsebine, kot je na primer senčenje strukture elementov HTML [27].

1.2 Metodologija

Na začetku bo opredeljen model brskalnika in groženj ter pregled različnih pristopov. Vsak pristop bo predstavljen skozi vidike trojice CIA skupaj s pomanjkljivostmi, ki jih posamezen pristop ima.

Iz ugotovitev posameznih pristopov, predvsem objektno zmožnostnega modela, bo zasnovana knjižnica, ki bo združevala ideje preteklih del na tem področju in novitet spletnega okolja. Mehanizmi knjižnice bodo osnovani na podlagi identificiranih občutljivih funkcionalnosti, ki jih pogosto zlorabljajo napadi XSS. Protiukrepi bodo sestavljeni iz mehanizmov, ki bodo aktivirali zaščitne politike - funkcije JavaScript, ki omogočajo dinamično zasnovo kontrol.

Pripravljeni zaščitni mehanizmi in testne politike bodo testirane za vpliv na hitrost delovanja spletne strani. Funkcije in konstrukti zamenjani s kontrolnimi mehanizmi bodo primerjani z originalnimi funkcionalnostmi glede na čas izvajanja. Tako bomo dobili kvantitativne ocene glede upočasnitve posameznih funkcij.

1.3 Struktura naloge

Magistrsko delo je sestavljeno iz štirih poglavij. Najprej je bil predstavljen problem in potreba po ukrepih za uporabo dodatnih varnostnih kontrol.

V drugem poglavju bo definirano okolje brskalnika in vsebine ter model groženj, ki predstavlja izpostavljenost primarne vsebine proti drugim vsebinam. Nato bo predstavljena trojica načel CIA, s pomočjo katerih bomo predstavili ter analizirali obstoječe pristope k izolaciji in varovanju vsebine v brskalniku.

V tretjem poglavju bo na podlagi ugotovitev obstoječih tehnik in trenutnih zmožnosti okolja predstavljena zasnova za programsko knjižnico. Ideja za knjižnico izhaja iz objektno zmožnostnega modela ter pomanjkljivosti obstoječih pristopov. Osnova za implementacijo mehanizmov in varnostnih politik bodo identificirane občutljive funkcionalnosti, ki jih zlorabljajo napadi XSS. Predlagani mehanizmi in skupine politik bodo predstavljene ter na koncu testirane glede na vpliv na delovanje v primerjavi z originalnimi konstrukti. S tem bomo pokazali, da je knjižnica ne samo ne-invazivna glede na vsebino, temveč tudi ne potrebuje izdatnejših računskih virov.

Končne ugotovitve ter zaključki bodo predstavljeni v četrtem poglavju.

Poglavje 2

Varovanje kode v spletnem odjemalcu

2.1 Model spletnega odjemalca - brskalnika

Spletni odjemalec - brskalnik - je računalniški program, ki z danega spletnega naslova (ang. URL) pridobi, prikaže in upravlja spletno vsebino. Brskalnik je kompleksna programska oprema sestavljena iz osmih podsistemov, ki uporabniku omogočajo uporabo spletnih mest. Razumevanje sestave in delovanja brskalnika je osnova za razumevanje, kako se izvaja vsebina in ukrepov za varovanje kode v njem.

Podsistemi brskalnika so [28, 29]:

- **Uporabniški vmesnik** - del brskalnika, katerega vidi in upravlja uporabnik.
- **Navigacija** - del brskalnika, ki upravlja z URI viri in navigacijo spletne strani (naprej/nazaj).
- **Izrisovalnik (ang. Rendering engine)** - del brskalnika, ki prikazuje vsebino spletne strani glede na HTML/XML definicijo in oblikovna določila CSS.

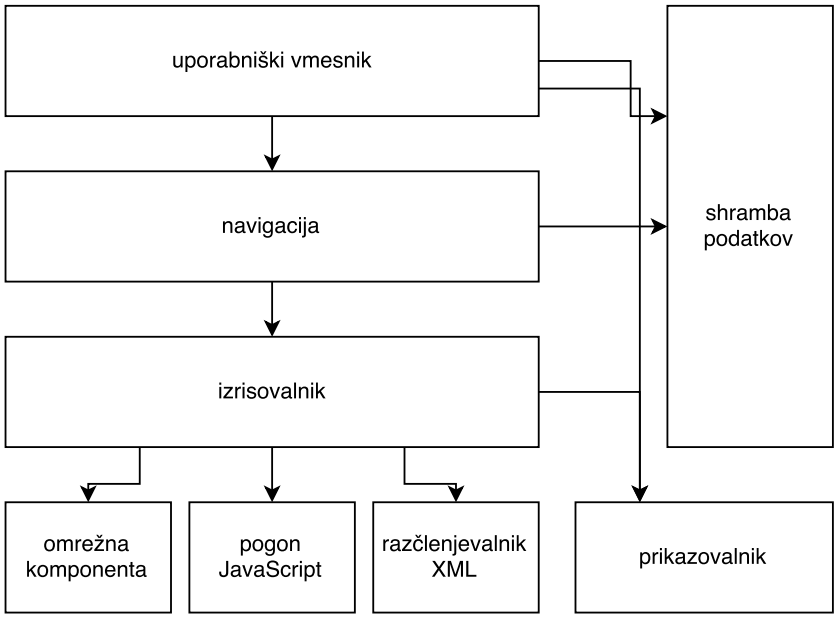
- **Omrežni sklop** - komunikacija preko spletnih protokolov, medpomnjenje in pretvarjanje med kodiranji.

- **Pogon JavaScript** - del brskalnika, ki izvaja programsko kodo spletne strani napisane v objektno orientiranem programskem jeziku JavaScript.

- **Razčlenjevalnik XML (ang. XML parser)** - razčlenjevalnik med definicijo vsebine XML in strukturo DOM. Kljub temu, da je dokument HTML podoben strukturi dokumenta XML, se njegova pretvorba izvaja kot del izrisovalnika.

- **Shramba podatkov** - shranjevanje podatkov, kot so zgodovina obiskanih naslovov, zaznamki, piškotki itd.

- **Prikazovalnik (ang. Display backend)** - prikaz uporabniškega vmesnika in spletne strani, kot jo je oblikoval izrisovalnik.



Slika 2.1: 8 podsistemov referenčne strukture brskalnika

Brskalniki sprejmejo vsebino v zapisih, ki jih določajo standardi spletnih tehnologij. Konzorcij W3C (World Wide Web Consortium) opredeljuje standarde:

- standard HTML za strukturo gradnikov spletne strani [4];
- skriptni jezik JavaScript za izvajanje programskih tokov [3];
- specifikacija CSS, ki določa nabor vizualnih lastnosti elementov HTML ter njihovo vedenje [4].

Postopek prikaza spletne strani se prične z določitvijo naslova URI izbranega spletnega mesta. V brskalnik se prenese datoteka HTML in prične se priprava prikaza spletne vsebine. Postopek prikaza je sestavljen iz treh faz:

- datoteka HTML se prenese v brskalnik in elementi HTML se zaporedoma pretvorijo v strukturo DOM;
- Pravokotniki, ki predstavljajo elemente se razporedijo, kot je določeno z določili CSS;
- Predstavitve elementov se izrišejo na zaslon s pomočjo prikazovalnika.

Koda JavaScript se izvaja med samim pretvarjanjem datotek HTML v strukturo DOM. Ko izrisovalnik prebere definicijo kode (značko HTML - "script"), ustavi pretvarjanje in prične izvajati kodo. Po zadnjem izvedenem programskem stavku se pretvarjanje nadaljuje.

Postopek prikaza spletnega mesta je postopen proces in se med nalaganjem virov večkrat izvede ter posodobi podoba spletne strani.

2.1.1 Programski vmesniki navideznega pogona JavaScript

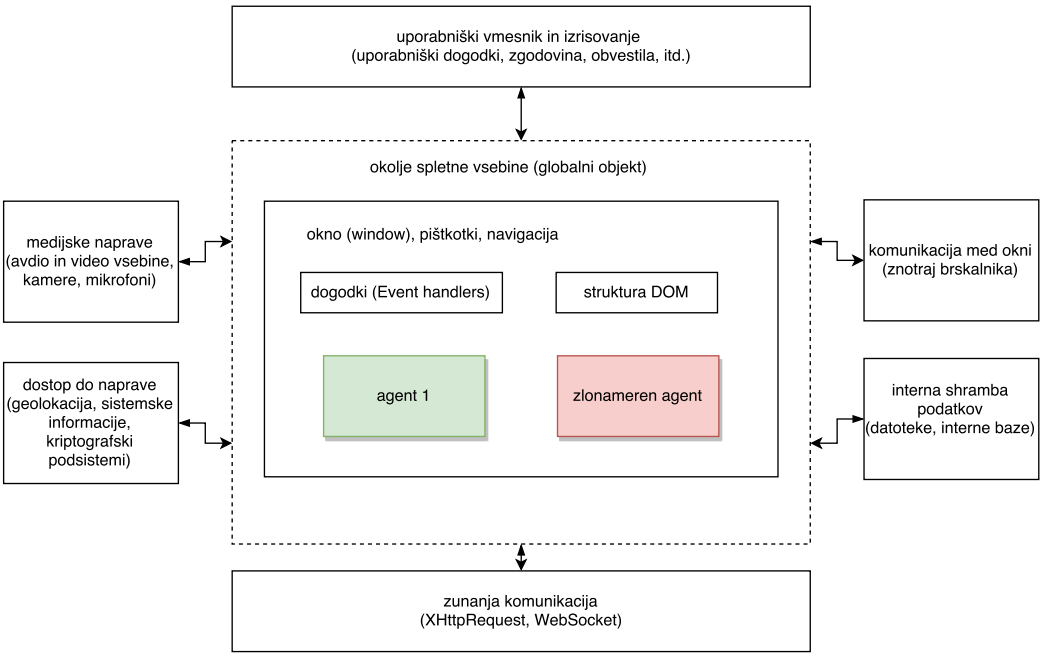
Zmožnosti jezika JavaScript so omejene z okoljem, ki mu ga ponuja brskalnik in obsega več programskih vmesnikov za manipulacijo s strukturo DOM, komunikacijo preko omrežja itd. Specifikacija HTML 5 omogoča kodi JavaScript dostop do obsežnih sistemskih zmožnosti gostujoče naprave. Vmesnike HTML 5 lahko razdelimo v sledeče kategorije [28]:

- komunikacija med odprtimi spletnimi okni v brskalniku ter tudi razširitvami;
- trajna hramba podatkov v napravi;
- zunanja komunikacija med različnimi viri (CORS) ter preko različnih protokolov (RTC, HTTP itd);
- podatki o napravi, kot so geolokacija, stanje baterije itd.;
- možnosti predvajanja avdio in video vsebin, dostop do medijskih naprav (mikrofon, video kamera);
- interaktivne zmožnosti, ki omogočajo odzivanje na uporabnikove klike, registriranje obvestil (ang. notifications), prehod v celozaslonski način itd.

2.1.2 Izvršljivo okolje in politika istih izvorov

Če bi imele spletne vsebine neomejen dostop znotraj brskalnika, bi lahko dostopale do vmesnikov in struktur DOM drugih spletnih strani. Brskalniki imajo zato za posamezne spletne aplikacije določeno izvršljivo okolje. Vsaka spletna aplikacija in okno (iframe) deluje znotraj meja, ki jih določa politika istega izvora (ang. Same-Origin Policy) [30].

Začetni dokument HTML brskalnik prenese z določenega naslova URL. Izvor (ang. origin) predstavlja ključne tri sestavne dele tega naslova - shemo



Slika 2.2: Spletna vsebina iz različnih virov (agenti) znotraj okolja in zmožnosti, ki jim jih ponuja brskalnik.

ime gostitelja ter vrata, s katerimi se določi vir dane vsebine (npr. datoteke HTML). Izvor datoteke z naslova URL "https:/fri.uni-lj.si:443" je shema "https", gostitelj "fri.uni-lj.si" ter vrata "443". V kolikor se dva vira razlikujeta vsaj v enem izmed treh sklopov, spadata med različne izvore (ang. different origins). Če sta vira prenesena z iste trojice, se privzame, da sta istega izvora (ang. same origin).

Brskalniki glede na različne izvore vsebine ločijo tudi funkcionalnosti, ki so posameznim virom na voljo.

Ločene funkcionalnosti za vsak izvor so:

- struktura DOM;
- shramba podatkov (spletna hramba, datotečni sistem itd.);
- pravice za dostop do posebnih funkcionalnosti naprave (geolokacija, celozaslonski način, medijske naprave itd.);

- pretočni viri WebRTC;
- zmožnost nalaganja virov iz istega izvora;
- zmožnost pošiljanja klicev AJAX brez omejitev.

Politika istih izvorov določa, da si viri iz istega izvora delijo izvršljivo okolje. Kadar želimo vključiti zunanje vire, nam mora tuja stran to dovoliti. Politika deljenja virov med izvori (ang. CORS policy) omejuje vključevanje tujih virov, ki nam niso dovoljeni. Dovoljenje je potrebno za dostopanje do drugih virov preko pristopa AJAX, vključevanje tujih pisav, slik in stilov ter predvsem programske kode. Dovoljenje se nahaja v glavi zahtevanega vira, ki vsebuje navedene izvore, katerim dovoljuje uporabo vsebine [31].

Politiki istega izvora in deljenja virov sta del specifikacij vseh brskalnikov. Nevarno zasnovana spletna vsebina pa lahko kljub danim omejitvam omogoči tuji zlonamerni kodi, da se prenese in izvede znotraj istega izvora in izvršljivega okolja [8]. Veliko spletnih strani uporablja dodatne knjižnice iz javnih virov, ki dovoljujejo, da se le te uporabljajo kjerkoli [32]. Tako dodana knjižnica se po prenosu v brskalnik prične izvajati v izvoru iz katerega je bila dodana. V primeru, da knjižnica vsebuje zlonamerno vsebino, ima poln dostop do strukture DOM in okolja. Podobno imajo možnost tudi dodatki za brskalnike, da vstavijo svojo vsebino v spletno stran in dostopajo do istih podatkov kot spletna stran, ki se prikazuje [7].

2.2 Model groženj

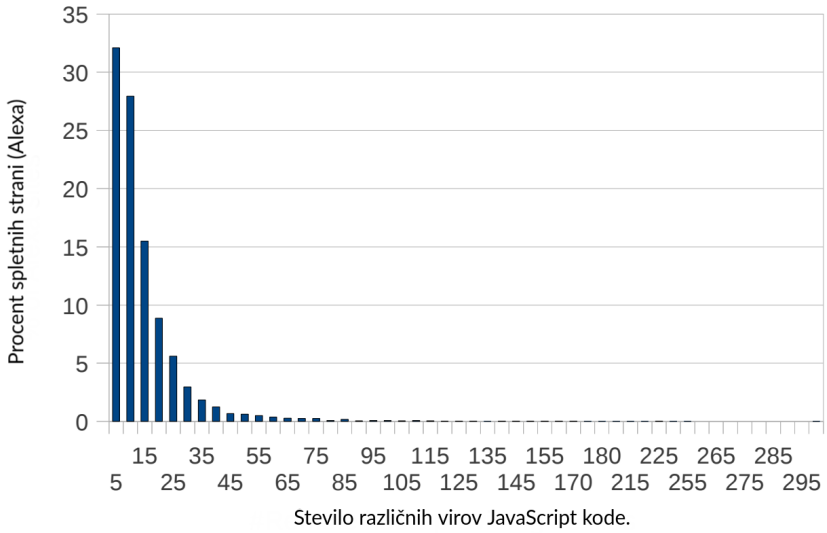
Skozi leta se je razvoj računalniških sistemov razvil iz monolitnih, centraliziranih računskih naprav v kompleksnejše sheme, kot so okolja strežnik-odjemalec ter raznorazne oblike distribuiranega računanja [33]. Distribucija kode in oddaljeno izvajanje je bila pomembna od samih začetkov povezovanja računalnikov v omrežja [34]. Sprva le skozi razvoj vtičnikov ter kasneje z razširitvijo spleta in spletnih tehnologij.

Spletne tehnologije so na prelomu stoletja sledile modelu strežnik-odjemalec, vendar je ob razvoju tehnologije in storitev ta model postal bolj osredotočen na samo končno napravo, pri čemer se je število zalednih sistemov povečalo oziroma so odjemalci dobili možnost komunikacije med samimi sabo [35, 36] ter večji obseg funkcionalnosti, ki jih lahko izvajajo [4, 35, 36].

Z varnostnega vidika je pomembno, da spletni ekosistem obravnavamo celostno. Kot je znano, je bil v shemi strežnik-odjemalec sprva nabor operacij in podatkov pretežno na strani strežnika, kar je posledično pomenilo tudi večje varnostno tveganje za uporabnike. Temu je sledil razvoj tehnik zaščite, ki se je osredotočal predvsem na zaščito strežniških virov ter zaščito komunikacijskega kanala [37]. Z večanjem procesne moči in zmožnosti končnih naprav (npr. nove zmožnosti kode po specifikaciji HTML 5 [4]) se funkcionalnosti sistemov selijo na končne naprave ter z njimi tudi tveganja.

Standard HTML omogoča, da spletna stran prikazuje in izvaja vsebino iz različnih virov [38], ki se na odjemalcu izvaja enakovredno oziroma so vsi elementi HTML in globalni objekti dostopni vsem vsebinam (isto izvršljivo okolje [30]). Delno se da nezaželeni vsebini preprečiti izvajanje s pomočjo politik [22], vendar brskalniki omogočajo dodatne kanale (vtičniki, razvojne konzole). Včasih želimo tuje vire namerno dodati na spletno stran. V tem primeru obstaja nekaj pristopov, ki jih standardi omogočajo in jih brskalniki trenutno podpirajo.

V grobem lahko izvajanje kode iz različnih izvorov v brskalniku opišemo z različnimi modeli, vendar bo dovolj, če opišemo z uporabo koncepta platforme - agentov [39]. Agenti so v tem kontekstu vsebine, ki prihajajo iz iste



Slika 2.3: Odstotek spletnih strani glede na število različnih virov JavaScript kode [32].

vsebinske domene - naj si bo z istega strežnika ali z različnih domen - ter je njihov namen odobren s strani primarne domene. Dodatni agenti so lahko posledica nenadzorovane definicije vsebin (slika 2.4), namerna vključitev dodatnih vsebin (ang. third party content), vsebina, ki jo vstavijo v stran vtičniki na brskalniku ali uporabniki preko razvojne konzole ali pa je posledica vdora nezaželene programske opreme [8].

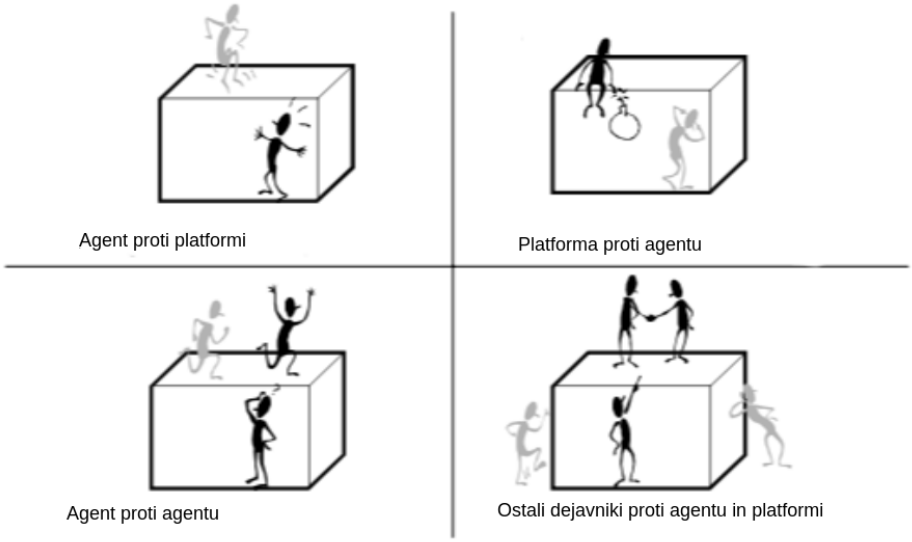
Na sliki 2.2 je predstavljen poenostavljen model brskalnika kot platforme. Agenti se lahko izvajajo tudi ločeno od glavnega okna kot vtičniki. Večina brskalnikov opušča podporo domorodnim vtičnikom ter se osredotoča na podporo standardom, kot je WebExtensions [40]. Slednji omogočajo izdelavo vtičnikov s standardi za razvoj spletnih vsebin. Standard predpisuje, da vtičnik izvaja svojo logiko ločeno od okna brskalnika (ang. background script) ali pa predpiše vsebino (content script), ki se vstavi v okno z vsebino spletne strani. S tem se sicer ne izvaja koda v istem izvajalnem kontekstu kot primarni agenti s strani razvijalcev platform, ampak si agenti delijo dostop do strukture DOM in elementov HTML ter nekateri brskalniki celo omogočajo

dostop do objektov v oknu in v izvornem vtičniku.

Iz danega modela na sliki 2.4 lahko predpostavimo sledeča tveganja [39]:

- napad agenta na platformo;
- napad platforme na agente;
- napad zunanjih dejavnikov na agenta v platformi;
- napad agenta na drugega agenta.

Varnost platforme ter njeno pravilno in varno izvajanje je v domeni gostujočega okolja in razvijalcev platforme, naj si bo to operacijski sistem ali vmesne platforme, ki zagotavljajo ista merila varovanja in izolacije programske opreme in procesov.



Slika 2.4: Model tveganj za brskalnik in spletno vsebino.

Spletni standardi zagotavljajo varnost vsebine na platformi skozi izvajanje vsebine v navideznem stroju. V tem okolju je vsebina in programski tok definiran s programskim jezikom JavaScript in deklaracijo vsebine (elementi

HTML), ki določajo prikaz vsebine. Dostop do sistemskih virov odjemalca ni mogoč neposredno, temveč je omogočen skozi definiran programski vmesnik (ang. application programming interface - API) [41, 40].

Podpisovanje in preverjanje avtentičnosti agentov se izvaja implicitno skozi varno komunikacijo in preverjanje avtentičnosti izvora vsebine na ravni komunikacije. [42]. Dodatna preverjanja so v domeni razvijalcev in brskalnikov.

Vprašanje pa se poraja pri možnostih, ki jih platforma omogoča za zaščito in izolacijo agentov v brskalniku. Prvotno se je smatralo, da razvijalec poskrbi za sestavo vsebine iz lastne ali tuje programske kode. Vendar se je izkazalo, da je ena ključnih varnostnih težav povezana ravno z nekontroliranimi izvajanji tuje programske kode (npr. napadi XSS). Po pregledu seznama OWASP deset najbolj pogostih in učinkovitih napadov se jih kar šest nanaša na izkoriščanje možnosti, ki jih ima lahko zlonamerna mobilna koda v našem brskalniku [43].

1. vstavitev in izvršitev poljubne kode;
2. ukradeni podatki za dostop;
3. nedovoljen dostop do podatkov in informacij;
4. razkritje zaupnih in občutljivih informacij;
5. zahtevek virov iz drugega izvora (napad CSRF);
6. neodobrene preusmeritve.

Vsakega izmed zgoraj naštetih napadov lahko izvršimo na spletnem vmesniku banke, v kolikor imamo dostop do vsebine znotraj okna brskalnika.

Napadu, ki ustreza zgornjemu opisu, pravimo lokacijsko prestopno skriptiranje oziroma napad XSS.

Napade XSS ločimo v tri kategorije glede na to, kako do same vstavitve tuje vsebine pride:

- shranjen napad XSS;
- preusmerjen napad XSS;
- napad XSS preko strukture DOM.

Shranjen napad XSS Do shranjenega napada XSS pride, ko je tuja vsebina shranjena na strežniku in se pošlje skupaj s primarno vsebino na brskalnik. Ta napad je predvsem pogost tam, kjer se shranjuje uporabniška vsebina ali pa ima spletni portal možnosti shranjevanja vnesene vsebine in le ta ni ustrezno preverjena.

Preusmerjen napad XSS Preusmerjen napad XSS se izvede podobno kot shranjen napad XSS. Namesto da je škodljiva koda posredovana iz zaupanja vrednega strežnika in izvršena, se pri tem napadu izvrši preusmeritev na ciljni portal skupaj s škodljivimi podatki. Ti podatki so lahko del obrazca ali naslova povezave in se na ciljni spletni strani zaradi neustreznega filtriranja vsebine vstavijo v okolje. Največkrat gre za neposreden prikaz podatkov poslanih preko parametrov POST ali GET. Če namesto imena v ustrezno polje obrazca vstavimo škodljivo kodo in se potem ta na ciljni strani vstavi v strukturo DOM ter izvrši imamo primer napada XSS.

Napad DOM XSS Napad DOM XSS je napad, kjer se vsebina vstavi v samo izvršljivo okolje preko vstopnih točk, ki jih elementi HTML omogočajo za izvršitev kode. To so običajno vstavljene "script" značke ali pa funkcije, ki se sprožijo ob dogodkih na teh elementih.

Vse tri vrste napadov XSS so med seboj prepletene. Na primer shranjen napad XSS je pravzaprav napad DOM XSS, le da je zlonamerna vsebina shranjena na zalednih sistemih. Enako velja za preusmerjen napad XSS, ki zlonamerno vsebino vsebuje v spletnem zahtevku. Seznam vseh identificiranih napadov DOM lahko najdemo na spletni strani `html5sec.org` [44].

2.3 Načela CIA

Vidik, s katerega bomo analizirali pristope k izolaciji in zaščiti agentov, sledi načelom in zahtevam trojice CIA.

Ti principi in zahteve trojice CIA so:

- zaupnost in avtorizacija podatkov ter funkcionalnosti,
- celovitost podatkov in funkcionalnosti,
- razpoložljivost podatkov ter funkcionalnosti.

Pri implikaciji vidikov trojice CIA je treba najprej opredeliti ključne pojme [12].

2.3.1 Zaupnost

Namen zaupnosti je omogočiti dostop do informacij samo tistim subjektom, katerim je bil dostop avtoriziran.

Vsa vsebina in ključne funkcionalnosti, ki sestavljajo mobilnega agenta, morajo biti zaupne oziroma morata biti uporaba in dostop avtorizirana. Morebitna nezaželena programska koda lahko nezavarovane podatke zlorabi. Z nezaščitenimi in neavtoriziranimi funkcionalnostmi lahko škodljiva koda operira izven predvidenih okvirjev delovanja ter onemogoči uporabo prvotne storitve ali pa izvede operacije, ki jih uporabnik ne želi ali pa ga celo oškodujejo [45]. Primer bi lahko bila škodljiva vsebina spletne banke, ki beleži vpisana gesla in avtorizacijske kode ter ob primernem času simulira interakcijo uporabnika po spletnem vmesniku za izvedbo nezaželene denarne transakcije. Podobne primere je bilo opaziti v preteklosti predvsem na spletnih družbenih platformah, kjer so napadalci vstavili škodljivo kodo v svoje predstavitvene strani ter zlorabili funkcionalnosti portalov v imenu uporabnika [8].

Zaupnost se zagotavlja skozi onemogočanje dostopa do občutljivih informacij oziroma funkcionalnosti. Pristopi k obvarovanju dostopa obsegajo različne načine, ki vključujejo [12]:

- določitev zaupnih podatkov in funkcionalnosti,
- varno hrambo,
- vzpostavitev avtoriziranega dostopa do podatkov in funkcionalnosti.

Dodatno bi lahko dodali še beleženje dostopov, vendar je to vprašanje odvisno od zahtev sistema in zaupnosti podatkov [12].

2.3.2 Celovitost

Celovitost je zmožnost zagotavljanja popolnosti in nespremenjenosti. Vsebina in funkcionalnosti potrebujejo celovitost. Celovitost vsebine je ogrožena, ko je izpostavljena spremembam in potencialnim manipulacijam. Zagotavljanje nezaželenih sprememb je mogoče zagotoviti najprej z omejevanjem dostopa ter zmanjšanjem možnosti sprememb. V izpostavljenih stanjih obstajajo različni pristopi k zagotavljajo celovitosti podatkov in stanja informacije ali funkcionalnosti (izvlečki, preverjanje podpisa funkcij, dodatna preverjanja itd.) [12].

2.3.3 Razpoložljivost

Razpoložljivost informacije ali funkcionalnosti omogoča avtoriziranim uporabnikom uporabo le te v predvideni obliki, brez nepredvidenih sprememb in modifikacij, ki lahko vplivajo na rezultat uporabe.

Platforma in agenti bi morali biti sposobni zaznati napake in spremembe, ki vplivajo na integriteto mobilnega agenta in pripadajočih vsebin. Obenem bi morali omogočati določeno mero odpornosti na napake oziroma ponuditi možnost obnovitve iz nastalih napak. Platforma mora tudi obenem zagotoviti sredstva in kvaliteto delovanja, ki jo agenti potrebujejo, oziroma obvestiti agente, da je prišlo do upada ravni kakovosti zagotovljenih storitev. Predvsem v primeru sobivanja agentov je v domeni platforme, da poskrbi za ustrezno razporeditev virov med agenti [12].

Zagotavljanje zaupnosti in integritete običajno vpliva na razpoložljivost in kakovost uporabe virov, predvsem računskih. Zaščita in dodatna preverjanja vsebine lahko povzročijo zamude v izvajanju programskega toka agenta, kar lahko vpliva na njegovo uspešnost izvajanja. Zagotovitev vseh treh principov trojice CIA tako zahteva dober premislek, kaj agenti potrebujejo, ter uskladitev zahtevnosti zaščitnih ukrepov in granularnost varnostnih politik.

2.4 Izolacija in varovanje vsebine na spletnem odjemalcu

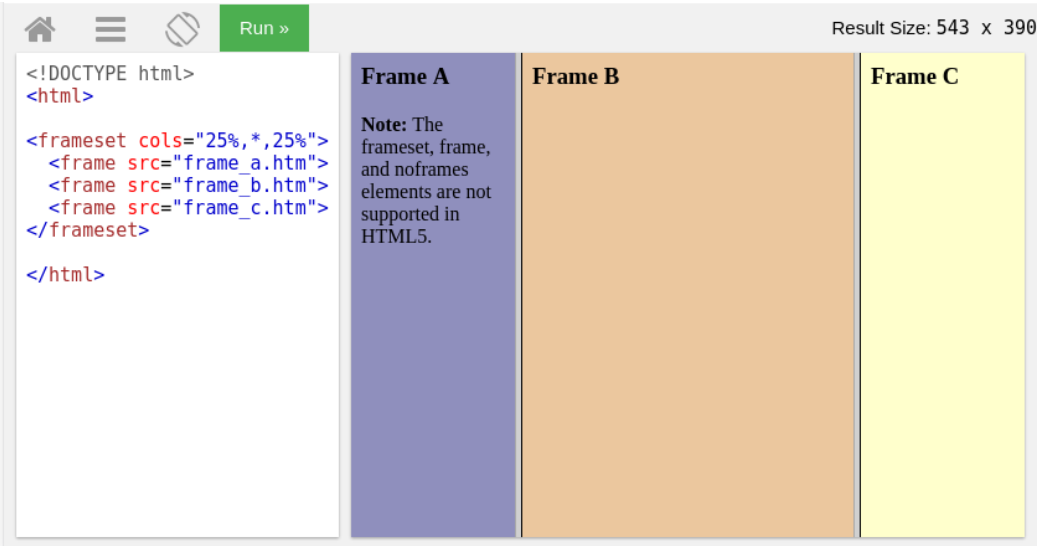
Vsebina spletnih strani je zasnovana na zanimivem izvršitvenem modelu, kjer aktivna vsebina - koda (skripti) v jeziku JavaScript ustvarjajo in spreminjajo obstoječo zasnovo strani ter lahko dodajajo nove funkcionalnosti med izvajanjem. Ta lastnost uvršča JavaScript med višje nivojske jezike, ki lahko modificirajo svoje izvajanje. Visoko nivojski jeziki omogočajo lažje izražanje razvijalcev, obenem pa odpirajo vrsto novih možnosti za napade, ki jih je težko zaznati in zaustaviti [24].

Obstaja nekaj pristopov, kako vsebino različnih virov združiti v končen produkt - spletni vmesnik. Nekatere med njimi so del obstoječih specifikacij ali pa različni pristopi k reševanju problematike z funkcionalnostmi, ki so na voljo. Spletne tehnologije so vodilni primer izvajanja mobilne kode in stalno v razvoju. Vsako leto se spletnim vsebinam dodajajo nove funkcionalnosti in nadgrajujejo obstoječe [13]. Nekatere nadgradnje so omogočile nove pristope k reševanju varnostnih problemov v brskalnikih [10, 23]. Medtem pa so marsikatere nadgradnje in ideje odprle vrsto novih možnosti za varnostne napade in izkoriščanje funkcionalnosti v škodljive namene.

V sledečih podpoglavjih so predstavljeni pristopi, ki omogočajo izboljšanje varnosti spletne vsebine in njeno izolacijo. Nekateri so standardizirani [46] ali pa so primeri uporabe obstoječih zmožnosti okolja in vsebine strani ter inovativni pristopi k celostnem reševanju izolacije in varovanja kode spletnega portala.

2.4.1 Okvirji (angl. frames)

Okvir (ang. frame) je del standarda HTML že od samih začetkov [47]. Prvotni namen okvirjev je bil razdelitev spletne strani na neodvisne razdelke, kot so glava, noga, navigacija, itd. V času, ko je bila vloga strežnika izključno gostovanje statičnih datotek, je razvijalec pripravil več podstrani ter s pomočjo okvirjev povsod vključil referenco na datoteko HTML, ki je vsebovala enotno vsebino, na primer navigacijo.

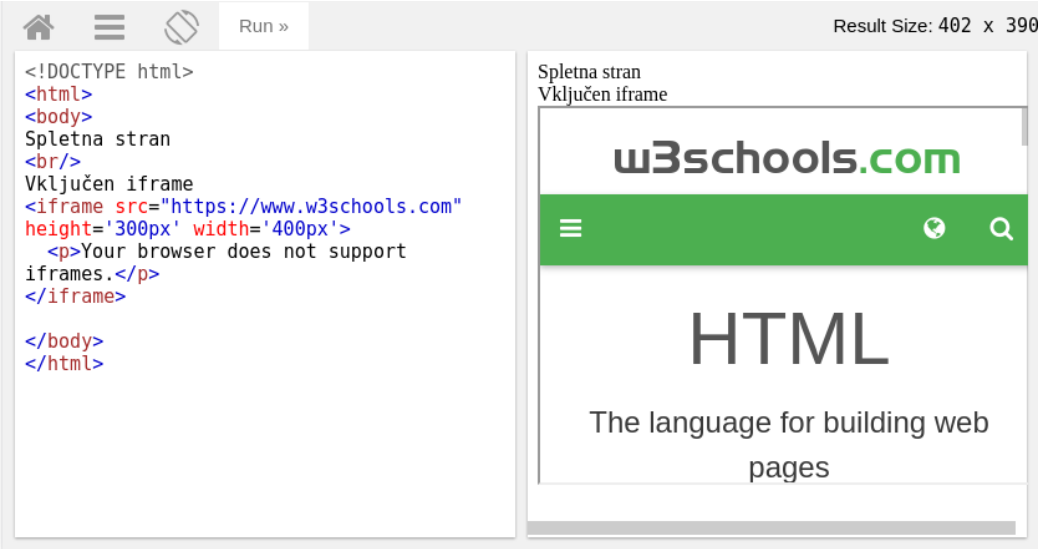


Slika 2.5: Razdelitev spletne strani z uporabo značk frameset in frame. Vsak razdelek je bil svoj vir vsebine - svoja datoteka HTML [48].

Prvotna uporaba okvirjev za postavitev spletne strani (slika 2.5) je bila nadgrajena s predstavitvijo HTML elementa IFRAME. Slednji je obdržal namen - vključitev zunanje vsebine, vendar so bile predstavljene nove zmožnosti in sicer vsebinska ločitev vključenih virov, s čimer se je dosegla izolacija vsebine. Oblikovno so bile nove značke fleksibilnejše, saj smo jim lahko določili poljubne dimenzije in gnezdenje v strukturi DOM [47].

Iframe značka je tudi postala osrednji pristop k vključevanju nezaupljive vsebine v spletno stran (slika 2.6). Pristop je bil uporabljen tudi za prikaz posebnih zunanjih funkcionalnosti na spletni strani, brez da bi oblikovanje

ali funkcionalnosti vplivale na vstavljen del ali gostujočo vsebino. To so uporabljala predvsem družbena omrežja za gumbe, ki jih je razvijalec lahko dodal na stran in so omogočali obiskovalcem, da so nemoteno sprožili izbrane operacije, na primer všečkanje ali deljenje strani.



Slika 2.6: Značka iframe je postala sinonim za vključitev izolirane tuje vsebine.

Z objavo standarda HTML 5 so bile značke frame in frameset ukinjene, značka iframe pa je dobila nove atribute, ki so omogočali več zmožnosti za kontroliranje vključene vsebine. Značka iframe vsebuje atribute, ki omogočajo nadzor varnostih vidikov vsebine kot na primer, ali lahko vključen dokument vsebuje forme ter odpira pojavna okna [4].

Atributi značke iframe so:

- **referrerpolicy** - politika nastavljanja polja priporočitelj HTTP (ang. Referrer), ki omogoča strežniku, da preveri izvorno stran zahtevkov HTTP. Polje je opcijsko.

    1. no-referrer - brez nastavitve (preprečuje pošiljanja naslova izvorne strani),

2. no-referrer-when-downgrade - če je stran povezana preko varne povezave (HTTPS), potem na vključene vire, ki niso dostopani preko varne povezave, ne pošiljaj naslova izvorne strani,
3. origin - pošlji samo domeno izvorne strani,
4. origin-when-cross-origin - pošlji polni naslov URL, če je vsebina iz istega vira, za zahteve na drugih straneh pošlji samo domeno izvorne strani,
5. unsafe-url - vedno pošlji polni naslov URL izvorne strani (brez parametrov).

- **sandbox** - določanje karakteristik izolacije kode:

1. allow-forms - dovoli forme in pošiljanje form,
2. allow-modals - dovoli obvestila (potrditev in opozorilno okno),
3. allow-orientation-lock - dovoli zaklepanje postavitve zaslona,
4. allow-pointer-lock - dovoli dostop do surovih podatkov o premiku miške in kazalnih naprav,
5. allow-popups - dovoli odpiranje novih oken,
6. allow-popups-to-escape-sandbox - dovoli odpiranje novih oken brez omejitev, ki so nastavljene za ta okvir,
7. allow-presentation - dovoli zaznavo predstavitvenega načina,
8. allow-same-origin - dovoli, da se vsebina privzame, da je iz istega vira [30],
9. allow-scripts - dovoli izvajanje JavaScript kode,
10. allow-top-navigation - dovoli dostop do navigacije spletne strani,
11. allow-top-navigation-by-user-activation - dovoli navigacijo samo za akcije, ki jih je sprožila interakcija;

- **srcdoc** - določitev vsebine lokalno preko kode JavaScript;

- **remote** - izvajaje vsebine v ločenem sistemskem procesu

Okvirji omogočajo izolacijo in nadzor nad zmožnostmi vsebine znotraj njih. Z onemogočenim dostopom do strukture spletne strani in konteksta JavaScript je dobro poskrbljeno za zaupnost vsebine, vendar je še vedno nekaj točk, ki jih brskalnik ne omejuje. Ena izmed njih je dostop do globalnega okna - window. S tem dostopom ne ogrožamo integritete ali razpoložljivosti ostalih strani, je pa mogoče priti do informacij o sistemu in brskalniku. Ta uhajanja podatkov so bila vodilo pri pripravi atributa *referrerpolicy*, saj lahko naslov glavne strani vsebuje tudi parametre, ki razkrivajo dodatne podatke o uporabniku.

2.4.2 Varnostna politika vsebine

Varnostna politika vsebine (ang. Content Security Policy) [46] je bila vzpostavljena z namenom reševanja težav, ki so nastale zaradi nekontrolirane vsebine spletne strani iz različnih domen in napadov, ki so to izkoriščali (napadi XSS in CSRF) [46, 49]. V času priprave specifikacije je ostajala samo politika SOP (ang. Same-Origin policy [30]), ki je omogočala omejitev vsebine na določen vir. Posamezen vir je določen s protokolom, domeno in vrati TCP spletnega strežnika [30].

Cilj politike CSP je bil zagotoviti nadzor nad viri, ki se lahko izvršijo v pripadajočem oknu spletne strani, zagotoviti višjo raven zaščite proti napadom, kot je napad XSS, ter vse skupaj zagotoviti opcijsko in s čim manjšimi zahtevami.

CSP politiko verzije 3 se vklopi in definira z meta značko Content-Security-Policy v glavi dokumenta HTML [22], v kolikor jo ciljni brskalnik podpira. Politika definira dve temeljni pravili - prepovedano umestitev kode kot vsebine ali atributa elementov HTML (značke script ter dogodkovne funkcije) ter prepovedano pretvorbo oziroma izvršitev kode iz besedila (funkcije kot so "eval" itd.).

Varnostno politiko je mogoče dodatno definirati preko direktiv. Za vsako

direktivo lahko navedemo veljavne vire vsebine oziroma uporabimo katero izmed rezerviranih vrednosti: none - noben vir, 'self' - vir trenutne strani.

- default-src - privzeta direktiva; vse nedefinirane direktive dobijo vire določene s to direktivo;
- base-uri - viri dovoljeni za base značko;
- block-all-mixed-content - kadar je stran naložena preko protokola HTTPS, prepreči nalaganje virov preko protokola HTTP;
- child-src - dovoljeni viri za konstrukte WebWorker in značke frame ter iframe;
- connect-src - dovoljeni viri, ki jih lahko uporablja aktivna vsebina (AJAX, fetch, a značka s ping atributom);
- font-src - dovoljeni viri za pisave;
- form-action - dovoljeni cilji za pošiljanje obrazcev;
- frame-ancestors - dovoljeni viri strani, ki lahko to stran vključijo v (i)frame značko;
- frame-src - dovoljeni viri frame značk;
- img-src - dovoljeni viri slik;
- manifest-src - dovoljeni viri manifest datoteke;
- media-src - dovoljeni viri medijskih značk (avdio, video);
- object-src - dovoljeni viri object značke;
- plugin-types - dovoljeni MIME tipi vtičnikov;
- report-uri - naslov, kamor najse posredujejo poročila o kršenju varnostne politike;

- sandbox - omogoča nastavitev sandbox atributov, ki se uporabljajo za iframe značko, na trenutni strani;
- script-src - dovoljeni viri skript;
- style-src - dovoljeni viri style značk;
- upgrade-insecure-requests - samodejno uporabi HTTPS protokol za vse klice na izvorni vir dane strani.

Večina direktiv je že podprta v najbolj razširjenih brskalnikih [22], nekaj pa jih je trenutno v pripravi:

- disown-opener - odstrani referenco na izvorno okno v pojavnih oknih.
- navigation-to - dovoljeni viri za navigacijo;
- referrer - dovoljeni viri strani, iz katerih je uporabnik lahko preusmerjen na dano stran;
- report-to - naslov, kamor naj se posredujejo poročila o kršenju varnostne politike oziroma proženje napake v izvajanju;
- require-sri-for - vse script in style značke morajo vsebovati atribut za integriteto;
- strict-dynamic - dovoli dinamično vstavljanje script značk iz kode iz dovoljenih virov;
- worker-src - dovoljeni viri skript za Web Workers.

Trenutno podprte direktive omogočajo nastavljanje virov vsebine in onemogočajo uhajanje podatkov preko nezaščitenih povezav (block-all-mixed-content, upgrade-insecure-requests). Ustrezno nastavljanje politike zna biti zahtevno, zato standard omogoča uporabo dodatne meta oznake "Content-Security-Policy-Report-Only", s katero lahko testiramo odziv brskalnika na predpisano vsebino spletne strani in določeno varnostno politiko.

Za razliko od uporabe iframe značk za izolirano vključevanje vsebine v ločenem kontekstu nam politike omogočajo, da uporabimo vsebino iz različnih virov, dokler le ta ustreza politiki izvora. Vsebina se nato izvaja v istem kontekstu in lahko dostopa do medsebojnih funkcij in DOM elementov.

Varnostne politike ne omogočajo večje granularnosti pri določanju omejitev glede posameznih delov strani. Težava postane izrazita, če upoštevamo, da večina najbolj obiskanih strani uporablja večje število skript iz deljenih virov, nekateri so med njimi bili že uspešno napadeni [6, 49]. Spremenjena koda, ki prihaja iz dovoljenega vira, lahko dostopa do vseh sredstev strani isto neomejeno kot predvidena vsebina iz izvornega vira. Natančnejše določanje omogočajo pristopi, kot so filtriranje vsebine ter namenske knjižnice, ki bazirajo na zmožnostih jezika in sredstev v brskalniku.

2.4.3 Celovitost virov

Celovitost virov [25] (ang. SubResource Integrity) je novost, ki omogoča razvijalcem, da poleg vključitve tuje kode, pisave ali stila dodajo kriptografski izvleček.

```
<script src="https://example.com/example-framework.js"
        integrity="sha384-oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYllkPzQho1wx4JwY8wC"
        crossorigin="anonymous"></script>
```

Slika 2.7: Primer vključitve vira z izvlečkom.

Na sliki 2.7 je primer vključitve kode iz dane domene. Izvleček na začetku vsebuje pripet naziv algoritma, s katerim je bil ustvarjen. Dodatno lahko tudi določimo, ali bomo pri zahtevku za vir poslali identifikacijske podatke, kot so piškotki, avtentikacijo ali certifikat [31].

Preverjanje integritete virov je priporočljivo predvsem pri vključitvi virov, ki jih vključujemo izven primarnega izvora. To so predvsem JavaScript knjižnice, običajno iz omrežij za distribucijo vsebin (CDN).

2.4.4 Filtriranje vsebine

Iframe značka omogoča umestitev tuje vsebine popolnoma ločeno od strani, kjer gostuje. Varnostne politike CSP po drugi strani dovoljujejo integracijo zunanje vsebine z vsebino spletne strani, vendar pri pogoju, da prihaja iz določenih virov.

Filtriranje vsebine (ang. content whitelisting) je pristop, ki je bil pogosto uporabljen pri spletnih portalih, ki so namensko želeli dovoliti tuji vsebini, da postane del primarne sestave ne glede na vir. Eden takih primerov so bile poosebljene strani na portalu Facebook, kjer so si avtorji strani lahko razvili lastno oblikovano pristajalno stran. Pristop, ki ga je ubral portal Facebook za izolacijo zunanje vsebine, je vključeval predhodno preverjanje skladnosti vsebine. Vsebina je morala biti skladna s FBML in FBJS specifikacijo, ki sta v bistvu samo podmnožica jezikov HTML in JavaScript brez možnosti izvajanja operacij, ki bi lahko škodile uporabnikom. Vsebina teh podstrani je bila shranjena na strežnikih portala Facebook, vse dokler tehnologija iframe značk ni dozorela za varno uporabo [50, 49].

Filtriranje spletne vsebine je preverjanje, ali vsebina vsebuje dovoljene elemente HTML in konstrukte jezika JavaScript. Slednja bi lahko vsebovala značke, ki bi omogočale vstavljanje vsebine (script, iframe itd.) ali izvršitev zlonamerne programske kode, ki lahko v uporabnikovem imenu izvede nezaželene operacije na vmesniku ali pa napadalcu posreduje občutljive informacije. Za vzpostavitev sistema je potrebno določiti, kolikšen obseg zmožnosti bomo dovolili. V primeru presega dovoljenih pravil lahko vsebino prečistimo ali pa jo zavrnemo ter ustrezno sporočimo neskladnost.

Filtriranje se izvaja, preden je vsebina naložena v brskalnik. Portali, ki shranjujejo uporabniško vsebino običajno vsebino prečistijo in jo le tako shranijo na strežnikih za kasnejši prikaz (npr Facebook [49]). Drugi način je nalaganje vsebine preko vmesnih sistemov. Spletna stran tem sistemom sporoči naslove vsebine, ki jo potrebuje, le ti pa jo najprej prečistijo in nato posredujejo (npr. Caja [10]). V primeru uporabe portalov, kjer je enotna točka dostopa do spleta (npr. ustanove, organizacije), je mogoča tudi name-

stitev vmesnih sistemov, ki vso vsebino prestrežejo in prečistijo - naj si bodo to namenski požarni zidovi ali pa samo dodatki za brskalnike.

V vsakem primeru je problem določiti, katera vsebina je obravnavana kot tuja ali kot ustrezna. Razvoj spletnih tehnologij je prinesel nove načine za izogib preverjanj sumljive kode [14, 16]. Dodatno težavo predstavljajo varne povezave, saj je povezava od brskalnika do strežnika zaščitena in je vmesni sistemi ne morejo nadzorovati.

Filtriranje vsebine je smiselno na portalih, ki želijo omogočiti tujo vsebino na svojem portalu. Primeri za to so bile prilagodljive Facebook strani ter tudi portali, ki so na primer omogočali poganjanje in testiranje lastne kode ter spletni portali, ki omogočajo objavo ustvarjenih besedil s strani uporabnikov.

Težave izvirajo iz določevanja, kaj je v določenem trenutku dovoljena vsebina ter razvoj novih zmožnosti vsebine. Včasih je določena funkcionalnost potrebna za uspešno delovanje vključene vsebine, vendar ji le ta omogoča zlorabo in zlonamerno delovanje. Preverjanje, kaj zmore tuja vsebina ob pogostih nadgradnjah specifikacij spletnih tehnologij, je večkrat naletelo na težave. Pogosto se omenja izogibanje sintaktičnim preverjanjem z zakrivanjem kode, uporabo pretvorbe besedila v izvršljivo vsebino, zlorabo tolerance brskalnikov do napačno predpisanih tipov vsebine ter zlorabo novih opcij brskalnika [14, 16].

2.4.5 Zmožnosti spletnih tehnologij

Do sedaj so bili našteti pristopi k zaščiti vsebine, ki omogočajo poganjanje določene vsebine v peskovniku ali onemogočajo nalaganje vsebin iz virov, ki niso dovoljeni. Filtriranje omogoča varnejši pristop, saj lahko mejo izolacije med primarno in sekundarno vsebino določimo natančno, vendar je za implementacijo potrebno preusmeriti vsebino skozi vmesno točko. V praksi to pomeni uporabo dodatne programske opreme na strežniku ali pa namestitev požarnih zidov ter vmesnih programov, ki prestrežejo vsebino.

Nekateri pristopi za filtriranje gredo še korak naprej. Razlogi so predvsem v razvoju novih funkcionalnosti spletnih tehnologij, ki omogočajo upo-

rabo novih programskih konstruktov ter izvajanja v brskalniku. Ob prelomu tisočletja so bili glavni standardi, ki so bili podprti s strani brskalnikov, plod uporabe takrat aktualnih tehnologij. Zapis HTML je bil zasnovan na podlagi notacije SGML [47], oblikovne značke CSS so bile prav tako del standarda HTML. Na podlagi specifikacije HTML je bil definiran tudi standard DOM, ki je definiral semantiko in manipuliranje z objektno predstavitvijo deklaracije HTML. Sam standard je bil osnova za deklaracijo strukture spletne strani.

Vse do verzije HTML 5 (2014) ni bilo predpisanega privzetega niti priporočenega jezika za izvajanje aktivne vsebine. Prvotna organizacija W3C je razvoj standarda nadaljevala v smeri jezika XML, kar pa ni bilo všeč skupini posameznikov, ki so oblikovali skupino WHATWG [51]. Slednja je želela razvijati jezik v smeri podpore novim funkcionalnostim in zmožnostim naprav, ki poganjajo brskalnik. Leta 2014 so predlagali specifikacijo HTML verzije 5, ki je prinesla vrsto novih zmožnosti, nove možnosti oblikovanja elementov HTML ter privzeto podporo jeziku ECMAScript (JavaScript). Slednje se lahko vidi na definiciji atributa "type" v znački script, ki je v verziji 5 postal opcijski z privzeto vrednostjo na "text/javascript" [4].

Standard ECMAScript verzije 3 in naprej je poleg programskih konstruktov predpisoval implementacijo globalnega objekta, ki poleg konstant služi tudi za dostop do funkcij brskalnika. API za implementacijo in manipuliranje s strukturo HTML je sledil standardu DOM, vendar način integracije ni bil določen. Kontekst ECMAScript je imel globalno spremenljivko, ki je bila referenca na izvorni element strukture DOM. S pripravo standarda Web IDL [40] organizacije W3C je bil definiran programski vmesnik za uporabo zmožnosti brskalnika in manipuliranja strukture DOM. Standard bazira na strukturi DOM, ki jo predpisuje standard HTML, ter na primerih uporabe v jeziku ECMAScript verzije 6 [40].

ECMAScript verzije 3 je bil desetletje vodilni jezik za implementacijo aktivne vsebine kljub temu, da so obstajale določene težave, predvsem pri interakciji z vmesnikom DOM na različnih brskalnikih, okrnjena podpora

novim funkcionalnostim, ki so jo posamezni brskalniki ponujali pod svojimi nazivi, ter težave z jezikom samim [52]. Verzija 5 je po dolgem času prinesla bolje definirane obsege programskih blokov in konstruktov ter nove funkcionalnosti, ki so omogočale upravljanje z lastnostmi objektov. Zanimive so predvsem nove funkcije za upravljanje s karakteristikami objektov in njihovimi lastnostmi na primer zamrznitev metod objekta, definiranje funkcij za uporabo in nastavitev lastnosti objektov (ang. getters and setters), definicija modulov ter striktnega načina, ki vklopi varnejši način delovanja programske kode [53].

ECMAScript 6 prinaša nekaj novih sintaktičnih okrajšav (angl. sytactic sugar, npr.: konstante) ter novih funkcionalnosti (razredi itd.). V tej verziji je tudi nekaj specifikacij, ki rešujejo varnostne težave, kot na primer spremenljivke z nastavljivim obsegom (let), redefinicija konteksta funkcij in konstrukta "this", nespremenljivi tipi (ang. immutable types) in definicija konstante, "Symbol" tip, ki omogoča anonimne lastnosti znotraj objektov, konstrukti "Proxy" in sposobnosti jezika za samo analizo (ang. Reflection) ter konstrukta WeakMap in WeakSet, ki preprečujeta uhajanje objektnih referenc [3].

Varovanje kode z zmožnostmi jezika JavaScript

Prototipna narava jezika JavaScript in funkcije za upravljanje z lastnostmi objektov so omogočile razvoj različnih pristopov za varovanje kode [54, 11, 10]. Ideja za varovanje sistema z zmožnostmi programskega jezika je znan pristop pri zasnovi operacijskih sistemov [10, 55]. Izhodišče ideje je, da koda vsebuje lastnosti, kot na primer nezmožnost ustvarjanje reference do programskih in sistemskih sredstev. Sistem podpira varne zmožnosti kode, če koda in procesi uporabljajo zmožnosti, do katerih so dobili neposreden dostop. Izolacija med dvema procesoma se doseže tako, da se jima dodelijo zmožnosti, ki se ne prekrivajo - nimata skupnih referenc, preko katerih bi lahko vplivala neposredno drug na drugega [10].

Objektno-zmožnostni model (ang. Object Capability) je adaptacija programskega jezika, da zagotavlja varne zmožnosti programskih konstruktov

[56, 9]. Taka zasnova programskega jezika je ekvivalentna implementaciji jezika, ki vsebuje preverjanja dostopnih pravic do sredstev [9]. Kadar varnih zmožnosti jezika ni mogoče doseči v popolnosti, je v namene izolacije izvajanja procesov dovolj doseči avtorizacijsko varnost.

Slednja opisuje dva koncepta dovoljene uporabe in dostopa do konstruktov. Avtorizacija je izpeljana iz prejšnje avtorizacije (ang. all access must derive from previous access), ki predstavlja, da če dva dela programov nimata preseka avtorizacij, sta med seboj izolirana. Obenem omogoča prenos avtorizacije med izbranimi konstrukti. Drugo načelo pravi, da skupek avtorizacij ne more presegati največje med njimi. S tem se zagotovi, da ni mogoče priti do dviga nivoja avtorizacije ter da ima dan konstrukt samo dane nivoje avtorizacije.

Programski jezik mora biti pomnilniško varen, da ustreza zmožnostnem modelu, in to JavaScript tudi je. Vendar JavaScript ni varen objektno-zmožnostni jezik, saj imajo vsi subjekti dostop do globalnih sredstev (spremenljivk) [54]. Zmožnostni sistem za programski jezik predpostavlja definiranje sredstev, subjektov, zmožnosti in avtoritete, ki jo zmožnost predstavlja. Sredstva so programski in sistemski konstrukti, ki jih uporablja subjekt v okviru zmožnosti, ki so mu dodeljene z dano avtoriteto [10].

Obstaja nekaj pristopov, ki izolacijo in varnost vsebin rešujejo skozi zmožnosti spletnih tehnologij:

- ConScript,
- AdSafe,
- BrowserShield,
- Google Caja,
- JSand,
- drugi pristopi [21, 1, 57, 58, 59].

ConScript

ConScript [23] je sistem, ki predpisuje uporabo politik za posamezne vire, nad katerimi želimo vzpostaviti avtorizacijo. Politike se izvedejo na prilagojenem odjemalcu, ki ima implementirane dodatne funkcije (advice). Politike se izvedejo, preden pride do prve izvršitve JavaScript kode. Same politike so zapisane v predlaganem atributu HTML elementa "script" in se izvedejo preden se izvede telo elementa.

Prvotno je predlaganih 17 politik, ki omejujejo uporabo najbolj kritičnih sredstev v brskalniku za izvedbo najpogostejših napadov (npr. XSS). Predlagano ogrodje omogoča dodajanje dodatnih poljubnih politik, ki jih brskalnik vzpostavi, preden preide v izvrševanje aktivne vsebine.

```
<head>
    <script policy='
        let httpOnly: K -> K = function(_ : K) {
            curse(); throw "HTTP-only piškotki";
        }
        around(getField(document, "cookie"), httpOnly);
        around(setField(document, "cookie"), httpOnly);'
    >
    </script>

</head>
```

Slika 2.8: Primer politike, ki preprečuje dostop do piškotkov spletne strani.

Neposredni testi so pokazali upočasnitve izvajanja kode za 3,42-krat ter 1,24-krat po optimizacijah kode. Ob testiranju praktičnega vpliva na delovanje in hitrost nalaganja strani so bile upočasnitve zanemarljive [23].

AdSafe

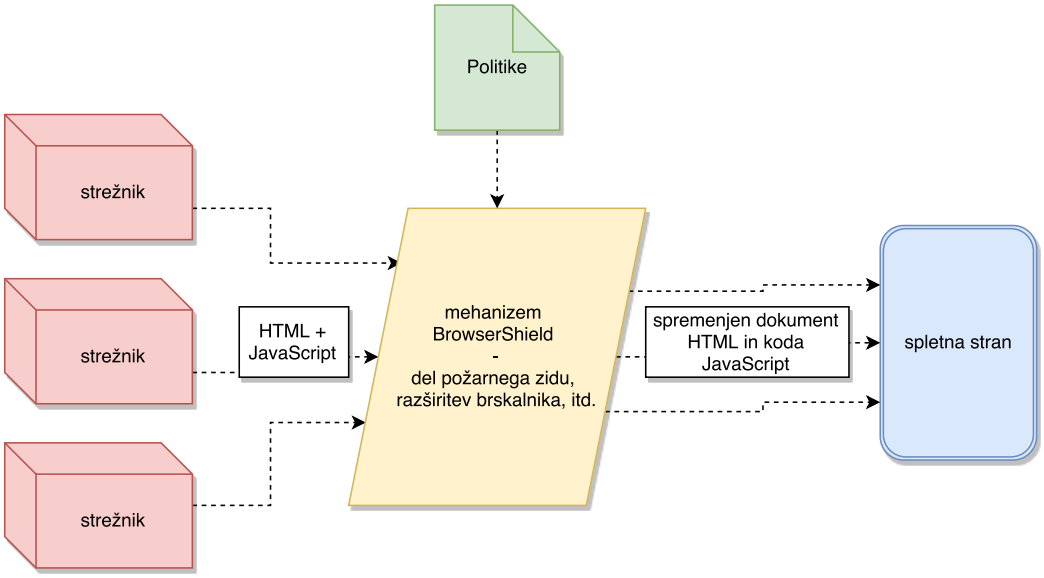
AdSafe orodje [6] je namenjeno vstavitvi tuje vsebine v dano spletno stran. Deluje tako, da izvede statično analizo vsebine ter preveri, da le ta ne vsebuje uporabe zmožnosti, ki so prepovedane za uporabo (npr. eval). Knjižnica potem izvede vsebino s klicem metode "ADSafe.go()". Slednja omogoči uporabo sistemskih sredstev skozi posredne konstrukte (ang. wrappers), ki skrbijo za ustrezen dostop in zmožnosti na izbranih sredstvih (prototipi objektov, globalni objekti in DOM elementi).

Ker preverjanje vsebine poteka pred prenosom v brskalnik, avtorji nava-

jajo, da ni opaznih upočasnitev izvajanja [60]. Nadaljnjo raziskovanje omenjene knjižnice je izpostavilo določene težave pri izolaciji dostopa do strukture DOM, možnostih za izvedbo napadov XSS, dostopanju do globalnega objekta ter spreminjanja prototipne verige [28].

BrowserShield

BrowserShield [26] ima podobno zasnovo kot AdSafe. Sestoji iz dveh faz. V prvi fazi je potrebno vso spletno vsebino prestreči ter preveriti, ali ustreza predpisanim pravilom. Nato se klice funkcij spremeni, da izvedejo funkcije z dodatnimi preverjanji. V drugi fazi se modificirana koda izvede na odjemalcu z dodatnimi preverjanji, ki skrbijo, da koda ne zlorablja danih zmožnosti sredstev [26].



Slika 2.9: Mehanizem BrowserShield deluje med izvorom in spletno stranjo in zamenja klice funkcij glede na politike.

Mehanizem BrowserShield deluje učinkovito, v kolikor lahko prestreže vire, ki jih spletna stran vključuje. To je lahko težava, če se datoteke prenašajo po zaščiteni povezavi [26]. Prav tako je mehanizem težaven, ker se pretvorba zgodi ob vsakem klicu na strežnik posebej, kar vpliva na čas, ki je

potreben za prenos virov do brskalnika. Testi so pokazali, da njegova uporaba v obliki požarnega zidu upočasni nalaganje strani tudi do trikrat [26, 28]. Zaradi preoblikovanja kode JavaScript, da se vsi klici izvedejo preko enotnega objekta "bshield", so bile izmerjene sto šestintridesetkratne upočasnitve izvajanja funkcij.

Google Caja

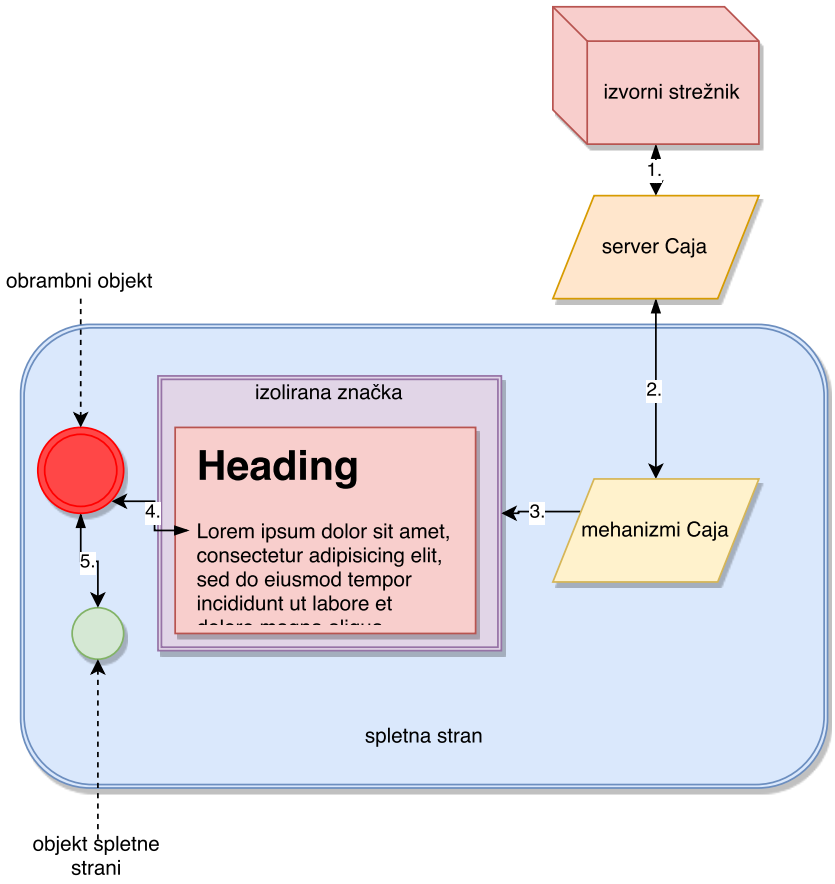
Google Caja podobno kot AdSafe in BrowserShield poskrbi najprej za statično analizo kode in skladnost vstavljene kode z standardom SES [61, 10]. SES je podmnožica jezika JavaScript z odstranjenimi varnostno občutljivi konstrukti. Preveri se tudi, da ni prostih spremenljivk in dostopa do referenc, ki bi lahko klicale funkcije izven izolacije. Slednje se preoblikuje, da se izvedejo preko vmesnikov. Sam postopek preoblikovanja vsebine je poimenovan "Cajoling" in se izvaja na vmesnem strežniku ter nato posreduje naprej k odjemalcu.

Caja obenem na samem brskalniku poskrbi za okrepitev okolja (ang. tempering), v katerem se izvaja. Prejeta izolirana koda se izvede znotraj JavaScript funkcije, ki omogoča dostop samo do določenih referenc in akcij. Tako vstavljena koda ne more spreminjati globalnih objektov ter vplivati na izvajanje ostale vsebine.

JSand

JSand [54] je v izvajanju podoben Google Caji s to razliko, da implementacija uporablja Proxy konstrukt za vzpostavitev membrane okoli vstavljene vsebine. Dodatno tudi preprečuje dinamično nalaganje kode z omejevanjem izolirane kode do operacij z DOM strukturo (vstavljanje značk script).

Zgoraj opisani pristopi vsebujejo podobne pristope k vzpostavitvi avtorizacijske varnosti. Vsi razen BrowserShield se osredotočajo na izolirano izvajanje vstavljene kode. Pristopi najprej poskrbijo za statično analizo vstavljene kode in odstranjevanje ali pa onemogočanje izvajanja operacij, ki lahko vplivajo na predvideno izvajanje spletne strani. Predstavljeni pristopi uporabljajo predpisane politike za določanje dovoljenih zmožnosti in avtorizacij



Slika 2.10: Primer delovanja sistema Google Caja za vključitev filtrirane zunanje vsebine.

vstavljene kode [54, 10, 26, 6, 23].

Poglavje 3

Varovanje vsebine z zmožnostmi spletnih tehnologij

Standardizacije in podpora novih spletnih tehnologij, ki jih podpirajo brskalniki [40], omogočajo nove pristope k varovanju in izolaciji vsebin spletnih strani. Nadgradnja jezika JavaScript ter poenotenje DOM objektne strukture med brskalniki omogoča nove pristope, obenem pa izpostavlja napake v obstoječih tehnikah.

V prejšnjem poglavju navedene tehnike za varovanje različnih virov vsebine opredeljujejo grobe razmejitve (ločen kontekst in DOM struktura pri okvirjih, globalno izločanje virov pri CSP, odstranjevanje nedovoljenih struktur skozi filtriranje itd). Alternativni pristop, ki se ga poslužujejo predvsem podjetja s spletnimi portali, ki gostijo neznano vsebino, vključuje filtriranje vsebine skladno z določenimi pravili. Ta pristop omogoča zlivanje spletnih vsebin, vendar se zahteva preusmeritev vsebine skozi vmesnik, ki jo prečisti. Razvoj spletnih jezikov tudi odpira vrsto novih možnosti za izogibanje takim ukrepom (npr. podpora unicode znakom za zapis vsebine).

Naprednejši pristopi z uporabo lastnosti jezika JavaScript in objektnih zmožnosti odpirajo vrsto novih možnosti za izolacijo vsebine ter tudi zaščito primarnih vsebin in izvajalnega okolja spletne vsebine. Veliko raziskovanja je bilo v smeri uporabe lastnosti spletnih tehnologij za določitev varnega po-

dobsega funkcionalnosti in izolacije le teh od globalnega okolja, v katerem se izvaja, ter primarne vsebine. Večine teh tehnik so zasnovane na specifikaciji EcmaScript verzije 3, ki pa je do sedaj dobila nekaj novih posodobitev ter se osredotoča na zasnovo in implementacijo objektno-zmožnostnega modela.

Zasnova predstavljenega pristopa se bo prav tako osredotočala na implementacijo objektno-zmožnostnega modela nad jezikom JavaScript, vendar novejše verzije (verzija 6 - ES6). Pristop se bo razlikoval v osredotočanju na predloge, ki so jih obstoječa raziskovanja navedla, da bi bila lahko izvedljiva v prihodnosti. Pokazano bo, da so omenjeni predlogi ob trenutno razvitih spletnih tehnologijah mogoči in učinkoviti ter omogočajo ne le izolacijo vsebin, ampak tudi tudi vzpostavitev definiranih varnostnih kontrol za izvršitev vseh vsebin znotraj seje brskalnika.

3.1 Nevarnosti in napadi

Pri pregledu tipičnih napadov, ki jih podaja organizacija OWASP [62], in primerov izkoriščanja funkcionalnosti HTML za napade XSS [44] lahko napade razdelimo v sledeče kategorije:

- vstavitev $<script>$ značke z nezaželeno vsebino;
- nastavitev funkcij za dogodke neposredno na elementih HTML (npr. onclick, onload itd);
- nastavitev vsebine pri posebnih značkah (object, embed), ki sprejemajo zakodirano vsebino base64, ki vsebuje elemente HTML;
- izkoriščanje nejasnosti pri razločevanju komentarjev in elementov v dokumentu.

Zadnji dve kategoriji sta samo načina, s katerima lahko pridemo do možnosti vstavitve izvršljive kode ali klicev zlonamernih funkcij. Nastavljanje funkcij za odziv na dogodke elementov HTML na samih elementih preko atributov se privzame za slabo prakso "nevsiljivega" JavaScripta (ang. Unobtrusive JavaScript) [63]. Slednje so ene izmed glavnih vstopnih točk za izvršitev kode v okolju ter varnostno slaba praksa [64].

3.1.1 Napadi na vidike CIA vsebine

Z napadi XSS se lahko pridobi popoln dostop do zmožnosti okolja JavaScript, ki se ga lahko uporabi za različne napade [64]. Napadalec lahko zlorabi podatke, ki so mu na voljo, izvede napade na druge portale ali izvede prenos zlonamerni programov. Eden takih primerov so bili izsiljevalski virusi, ki so s pomočjo JavaScript kode prenesli na računalnik (ang. drive-by-download) in okužili gostitelje.

```
<script>
    var data = "TvrDiddmn...";

    var cmd = "23lml3ml3m4lmmd..sdl3l434lsdm=";
    var key_cmd = "2c3d34c4...444";
    var dec_cmd = CryptoJS.AES.decrypt(cmd, key_cmd);
    dec_cmd = CryptoJS.enc.Utf8,stringify(dec_cmd);
    eval(dec_cmd);

</script>
```

Slika 3.1: Primer JavaScript kode, ki je izvedla prenos zlonamerne kode [65].

Napadi na zaupnost občutljive vsebine

- Profiliranje brskalnika,
- kraja piškotkov,
- zloraba podatkov iz strukture DOM (CSRF ključi, bančni podatki itd),
- klicanje nedovoljenih vmesnikov,
- sledenje pritiskom tipkovnice in premikom miške,
- izvajanje Ajax klicev.

Celovitost vsebine

- Spreminjanje strukture DOM,
  - vstavljanje zlonamerne vsebine,
  - spreminjanje podatkov na strani,
- spreminjanje funkcij okolja [28]

Dostopnost vsebine

- Preusmeritve uporabnika (napad XSS s preusmeritvijo),
- DoS napadi na druge strani [64],
- vpliv na delovanje strani ali gostitelja.

Prav tako velja za slabo prakso uporaba varnostno problematičnih funkcij kot so "eval", "setTimeout","setInterval", "new Function()" za izvedbo kode iz niza [64].

3.2 Objektno-zmožnostni model

V objektni paradigmi izvajanja programske kode so subjekti, ki izvajajo akcije nad objekti. Instanca objekta je zmes internega stanja in programske logike, kjer je stanje zbirka referenc na ostale objekte. Računski sistem je potemtakem dinamičen graf referenc med objekti, ki si preko referenc izmenjujejo sporočila [56].

Objektno-zmožnostni model uporabi graf referenc kot graf dostopov in zahteva, da se lahko objekti sporočila pošiljajo samo preko referenc, ki so mu dodeljene. Za prehod med objektnim modelom in objektno-zmožnostnim modelom je potrebno zagotoviti, da referenc ni mogoče ustvariti, da ni mogoče dostopati do internih stanj drugih objektov, da statičnih vrednosti ni mogoče spreminjati itd. [56]. Na primer funkcionalnost programskega jezika C++ za pretvorbo števil v pomnilniške naslove krši dana načela, zato jezik ustreza objektnemu modelu in ne objektno-zmožnostnemu.

Programski jezik mora biti pomnilniško varen, da ustreza zmožnostnemu modelu, in to JavaScript tudi je. Reference znotraj brskalnika so znotraj enotnega pomnilniškega prostora in v pomnilniško varnem okolju. Komunikacijski kanal preko takih referenc ima lastnosti, kot so [56]:

- zaupnost - sporočilo, ki si ga izmenjata dva objekta, je dosegljivo samo njima;

- celovitost - sporočilo in referenca sta nespremenljiva;
- nezmožnost ponaredbe - objekt ne more ustvariti reference in komunikacije na objekt, do katerega nima dostopa;
- pristnost - četudi ima objekt referenco do danega objekta, ne more prestreči in preusmeriti sporočila poslana na isto referenco iz drugih objektov.

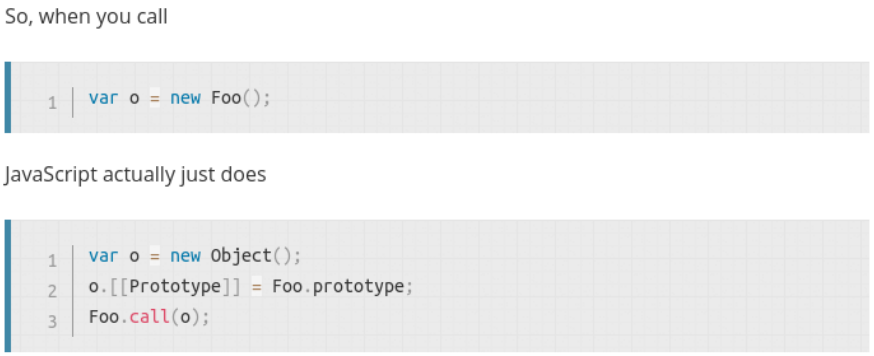
Za oblikovanje varovanja objektov preko njihovih zmožnosti je potrebno opredeliti globalne JavaScript objekte in njihove lastnosti skupaj z njihovimi določili, pri čemer je potrebno zaradi prototipnega dedovanja preverjati celotne verige prototipov.

3.2.1 JavaScript prototipno dedovanje

Posebnost jezika JavaScript v primerjavi z ostalimi objektno orientiranimi jeziki je med drugim v prototipnem načinu dedovanja. Jezik pozna šest primitivnih vrednosti in objekte. Tudi funkcije so zasnovane kot objekti s pripadajočimi lastnostmi in privzetimi metodami za izvajanje. Vsak objekt je pravzaprav seznam lastnosti z referenco na pripadajoče vrednosti (angl. mutable set of strings to values) [66].

Vsak objekt lahko deduje lastnosti in metode preko rezervirane lastnosti "prototip" (ang. prototype). Ob inicializaciji novega objekta iz konstruktorske funkcije se prototip lastnost uporabi za določitev starša pri dedovanju, ki se uporablja za izvrševanje interne logike iskanja dedovanih lastnosti. Ob inicializaciji objekta *(primer: new TargetObject())* se ustvari prazen objekt (seznam). Nastavi se mu starš v rezervirano lastnost "[[Prototype]]", katere vrednost zasede lastnost "prototype" iz prototipnega objekta *(TargetObject.prototype)*. Starš dedovanega objekta je dosegljiv tudi preko metode Object.getPrototypeOf ali preko lastnosti "__proto__", ki je sicer opcijska glede na brskalnik [3]. Ob klicu metode ali pridobivanju vrednosti lastnosti se izvede iskanje lastnosti najprej v začetnem objektu, nato v njegovem staršu

in naprej po verigi staršev, dokler se ne najde ustreznega zapisa lastnosti [66].



Slika 3.2: Inicializacija objekta JavaScript.

```
1  var a = {a: 1};
2  // a ---> Object.prototype ---> null
3
4  var b = Object.create(a);
5  // b ---> a ---> Object.prototype ---> null
6  console.log(b.a); // 1 (inherited)
7
8  var c = Object.create(b);
9  // c ---> b ---> a ---> Object.prototype ---> null
10
11 var d = Object.create(null);
12 // d ---> null
13 console.log(d.hasOwnProperty);
14 // undefined, because d doesn't inherit from Object.prototype
```

Slika 3.3: JavaScript dedovanje in iskanje lastnosti.

Prototipni način dedovanja v JavaScript jeziku je zelo dinamičen, saj omogoča spremembo prototipne verige in spreminjanje lastnosti staršev, s čimer vplivamo na vse ostale objekte, ki dedujejo iz njih [66].

3.2.2 JavaScript atributi lastnosti in objektov

Vsaka lastnost objekta ima atribute ali določila, ki opisujejo njene karakteristike. JavaScript definira dva tipa lastnosti - "podatkovne" (ang. data) in "dostopne" (ang. access) lastnosti.

Atributi podatkovne lastnosti so:

- **Value** - vrednost lastnosti.
- **Writable** - ali je lastnost mogoče prepisati.
- **Configurable** - ali je mogoče nastaviti atribute lastnosti.
- **Enumerable** - ali je lastnost vidna v seznam lastnosti objekta.

Atributi dostopne lastnosti so:

- **Get** - metoda, ki se izvede ob klicu lastnosti.
- **Set** - metoda, ki se izvede ob nastavljanju lastnosti.
- **Configurable** - ali je mogoče nastaviti atribute lastnosti.
- **Enumerable** - ali je lastnost vidna v seznamu lastnosti objekta.

Atribute lastnosti lahko nastavljamo s pomočjo metod Object.defineProperty(-ies), preko katerih lahko tudi lastnosti spreminjamo iz ene oblike v drugo. Spreminjanje atributov je mogoče, dokler je atribut "configurable" omogočen.

Brisanje lastnosti iz objektov je prav tako mogoče, dokler nam to omogoča "configurable" atribut.

Sam jezik omogoča, da lastnosti z vrednostmi nadomestimo s funkcijami, ki lahko izvedejo dodano logiko in vrnejo ustrezen rezultat, brez da bi morali spremeniti del kode, ki kliče lastnost.

Podobne atribute za omejevanje spreminjanja lahko nastavimo tudi nad samimi objekti. Objekt v osnovi nima omejitev glede dodajanja in spreminjanja lastnosti, vendar ponuja metode, s katerimi lahko spreminjanje objektov omejimo:

- **preventExtensions** - onemogočanje dodajanja novih lastnosti,
- **seal** - onemogočanje dodajanja lastnosti in spreminjanja prototipa,
- **freeze** - onemogočanje dodajanja in spreminjanja lastnosti ter spreminjanja prototipa.

Z zamenjavo podatkovnih lastnosti z dostopnimi lahko na poljubnem objektu vzpostavimo zmožnostni model. Dostop do pravih referenc se izvrši preko posrednih dostopnih funkcij, pri čemer obstoječe kode ni potrebno spreminjati kot v podobnih pristopih [26, 10]. Z nadaljnjim omejevanjem spreminjanja objektov preprečimo modifikacije dostopnih funkcij in spremembe vzpostavljenih preverjanj.

3.2.3 Izvršljivo okolje JavaScript, globalni objekt in strukture

Za vzpostavitev celostnega zmožnostnega modela nad objekti v spletnem kontekstu je potrebno opredeliti privzete objekte, njihove lastnosti z atributi ter pripadajoče prototipne verige.

Okolje JavaScript v brskalniku predpisuje globalni objekt, ki je ustvarjen iz prototipa Window. Globalni objekt vsebuje konstruktorske funkcije za večino objektov JavaScript, ki se uporabljajo za interakcijo s strukturo DOM in brskalnikom. Objektov JavaScript je preko štiristo petdeset (450) [67], ki so povezani med sabo v prototipno drevesno strukturo. Če odstranimo liste prototipnega drevesa dobimo dvainštirideset (42) osnovnih prototipov objektov, preko katerih se pri klicu lastnosti iščejo ustrezne reference 3.4.

Kot je bilo omenjeno, je potrebno za vzpostavitev zmožnostnega objektnega modela definirati reference ter vzpostaviti kontrole nad dostopom do le teh. Dostop do referenc v izvršljivem okolju je mogoč preko globalnega objekta in konteksta izvršitve (ang. Execution environment). Na primeru spletnega okolja reference v izvršljivem kontekstu izhajajo ravno iz definiranega globalnega okolja, ki je predstavljen z globalnim objektom in dosegljiv

preko lastnosti "this" v globalnem kontekstu ali pa preko lastnosti "window" danega izvršljivega konteksta [3].

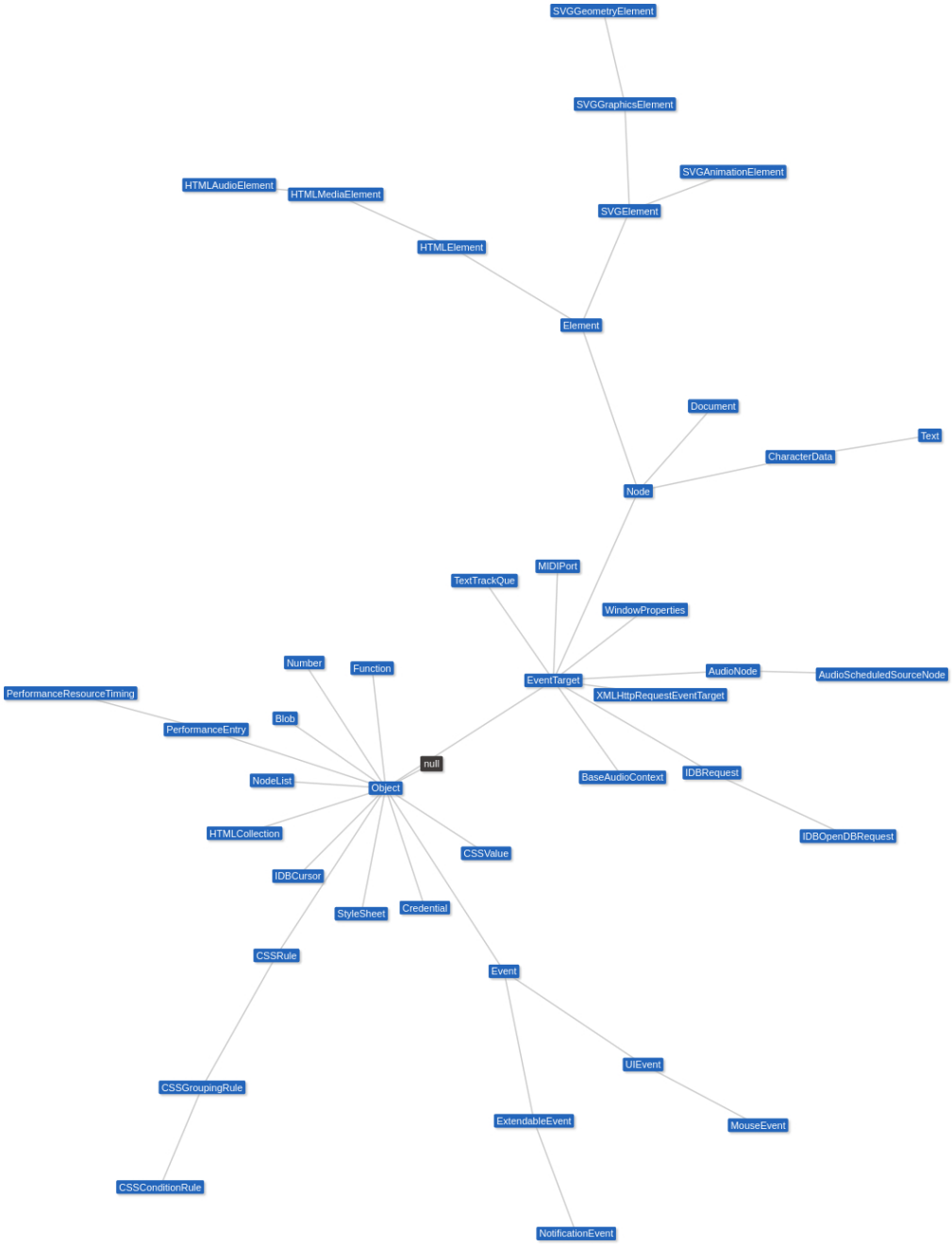
Globalni objekt v najbolj razširjenih brskalnikih ima preko sedemsto petdeset (750) lastnosti, izmed katerih je večina ali zamenljivih ali pa nastavljivih. To pomeni, da imajo omogočeni določili lastnosti 'writable" ali "configurable". Nekaj lastnosti pa ima določila, ki onemogočajo spreminjanje. To so lastnosti:

- "Infinity" - numerična vrednost za neskončnost;
- "NaN" - numerična vrednost za neustrezno številko;
- "undefined" - vrednost, ki označuje nedoločenost (običajno spremenljivke ali lastnosti);
- "window" - referenca na globalni objekt;
- "top" - referenca na globalni objekt okna, ki je najvišje v hierarhiji predvsem v primeru, da imamo podokna in okvirje;
- "location" - objekt za usklajevanje navigacije trenutnega okna;
- "document" - objekt za interakcijo s strukturo DOM v brskalniškem oknu.

"Infinity", "NaN" in "undefined" so vrednosti, ki jih uporablja JavaScript za interno delovanje [3]. Ostale lastnosti predstavljajo del vmesnika za komunikacijo z okoljem in brskalnikom.

"location" je lastnost, ki omogoča, da neposredno pridobimo ali nastavimo vir, ki ga prikazuje trenutno brskalniško okno. Ker je na določenih brskalnikih ni mogoče spremeniti, obstaja še možnost sprememb lastnosti in metod, ki jih vsebuje. Vendar pri pregledu lastnosti ugotovimo, da nobena izmed lastnosti nima omogočenega spreminjanja - za razliko od lastnosti "document", ki vsebuje "location" ter je objekt, ki deduje iz objekta "Document". Slednji ima vse lastnosti spremenljive, kar pomeni, da jih lahko

zamenjamo za funkcije, ki vsebujejo ustrezna preverjanja. Ostali lastnosti "window" in "top" sta zgolj referenci na globalni objekt.



Slika 3.4: Osnovni prototipi objektov JavaScript.

3.2.4 JavaScript posebnosti

Kontekst in doseg spremenljivk

V JavaScriptu poznamo dva različna dosega spremenljivk (ang. scope), v katerih se nahajajo spremenljivke - globalni in lokalni doseg. V globalnem dosegu se izvede koda, ko vstopi v brskalnik preko značk. Lokalni doseg spremenljivk je znotraj funkcije. Globalni doseg je vedno dosegljiv in vsebuje dostop do globalnega objekta in globalnih spremenljivk. Lokalni obseg je dosegljiv samo znotraj funkcije [3].

Poleg obsega pozna JavaScript tudi kontekst, ki je dostopen preko besede "this". Uporablja se predvsem pri metodah objektov in dedovanju, kjer se lahko prototipne metode navezujejo na objekt v kontekstu, le ta pa se ob inicializaciji dedičev nastavi na njih [3].

Uporaba lokalnega dosega pride uporabna pri gnezdenju funkcij, saj ima vsaka podfunkcija ob definiciji dostop do referenc in lokalnega dosega nadfunkcije. To nam omogoča, da lahko uporabimo funkcijo, ki lokalno ustvari reference in jih preda podfunkciji, le to pa vrne kot rezultat. Dostop do reference ni več mogoč razen preko vrnjene funkcije, ki pa le te nima v svojem lokalnem dosegu spremenljivk.

```
var test = function () {
    var value = 'SECRET TEXT'

    // nadomestne funkcije za dostopanje do vrednosti
    return function(app) { return value+app }
}

var test2 = test()
```

Slika 3.5: Primer skrivanja reference preko lokalnega dosega funkcije.

Zaprtja in moduli

Funkciji s slike 3.6 pravimo obseg učinkovanja ali zaprtje (angl. closure). Omogoča skrivanje referenc v lokalnem dosegu spremenljivk primarne funkcije. Dodatno lahko onemogočimo tudi dostop do same funkcije, in sicer z uporabo t. i. modulov. Funkcijo zaprtja definiramo kot anonimno funkcijo in jo neposredno izvedemo. Na ta način se lahko v jeziku JavaScript simulira zasebne in javne metode ter lastnosti objektov.

```
var Module = (function() {
    var privateProperty = 'foo';

    function privateMethod(args) {
        // do something
    }

    return {

        publicProperty: '',

        publicMethod: function(args) {
            // do something
        },

        privilegedMethod: function(args) {
            return privateMethod(args);
        }
    };
})();
```

Slika 3.6: Primer modula JavaScript.

Striktni način

Razvijalci JavaScript jezika so se zavedali težav, ki so nastajale zaradi konstruktov, kot so "with" in "this". Da ne bi povzročili prevelikih sprememb v jeziku, so omogočili uporabo striktnega načina (ang. strict mode). Aktivira se lahko v globalnem ali posameznem lokalnem dosegu spremenljivk [54].

Pretvorba niza v kodo

JavaScript je zelo dinamičen jezik, ki omogoča, da preko posebnih funkcij posamezen niz izvršimo kot programsko kodo. Vse funkcije razen "eval" kodo izvršijo v globalnem dosegu, "eval" pa ima sposobnost, da niz izvrši v lokalnem dosegu in kontekstu:

- setTimeout(niz),
- setInterval(niz),
- eval(niz),
- new Function(niz).

Asinhrono izvajanje

Koda v brskalniku se izvaja sekvenčno, torej en ukaz naenkrat. Vendar jezik JavaScript kljub temu omogoča asinhrono izvajanje kode, s čimer brskalnik optimizira izrabo sistemskih virov ter dogodkovno vodeno izvajanje (ang. event driven execution) [3].

Način asinhronega izvajanja je pomemben pri pripravi zaščitnih mehanizmov, saj ne more priti do tekme za vir (ang. race condition). Asinhrona koda se nastavi v izvršitveno vrsto, ki jo pogon JavaScript prične izvajati takoj, ko zaključi s prejšnjo kodo. Tak način izvajanja ne pozna funkcij, ki bi blokirale izvajanje (razen vmesnikov alert in confirm). Vsi dalj trajajoči konstrukti sprejemajo povratne funkcije, ki se sprožijo ob dokončanju interne logike.

3.3 Programska knjižnica

Ideja za implementacijo knjižnice, ki bi omogočala okrepitev varnosti primarne vsebine in uporabnika, temelji na dveh ključnih pristopih. Prvi je uporaba zmožnosti jezika JavaScript in virtualnega stroja v brskalnikih. Drugi je zagotovitev izboljšanja vidikov varnosti skozi implementacijo varnostnih politik, ki onemogočajo znane napade XSS [8, 44, 62] ter omogočajo razvijalcem omejitve dostopa do določenih funkcionalnosti in podatkov.

Prvi pristop izhaja iz lahkega samovarovanega JavaScripta (ang. Lightweight Self-Protecting JavaScript [11]). Pristop uporablja načela objektnih zmožnosti in prestrezanja dostopa do referenc s pomočjo principov aspektnega programiranja. Varovane funkcije se zamenja z vmesnimi funkcijami (ang. wrappers), ki upravljajo z dostopom do ciljnih funkcij [11].

Drugi pristop temelji na identifikaciji občutljivih funkcionalnosti spletnega okolja. Dober del teh zajema sedemnajst politik, ki jih opisuje ConScript [23]. Ostale smo identificirali skozi klasifikacijo funkcionalnosti glede na vidike CIA v razdelku 3.1. Naštete funkcije in funkcionalnosti ne zajemajo celotnega spektra vstopnih točk, ki jih napadalci izkoriščajo, temveč se osredotočajo na najpogostejše napade [8, 44]. Obenem je predstavljen pristop generične predstavitve obrambnih tehnik, ki jih lahko uporabimo na katerem koli izmed 450 objektov JavaScript, ali njihovih lastnosti, ki imajo ustrezno nastavljene atribute (atribut *configurable* ni enak "false").

Obe omenjeni deli, ki predstavljata osnovo za oba vidika implementacije imata nekaj ključnih pomanjkljivosti. ConScript je bil zasnovan kot nadgradnja brskalnika, ki namesto vmesnih funkcij kliče interno funkcijo *advice*. Brskalniki te nadgradnje ne vsebujejo, niti ni omenjena v sklopu Web IDL planov [40]. Pomanjkljivosti pri lahkem samovarovanem JavaScriptu so predvsem uporaba funkcij in funkcionalnosti, ki niso več podprte. Obenem pa ima knjižnica težave pri pomanjkljivostih vmesnih funkcij [68]. Omenjene težave in rešitve [68] bodo upoštevane pri zasnovi zaščitnih funkcij.

Občutljive funkcionalnosti bodo identificirane glede na najbolj razširjene brskalnike in njihovo podporo spletnim standardom. Vključenih bo trinajst

najbolj razširjenih brskalnikov glede na uporabo [69], ki presegajo 1% tržnega deleža. Brskalnik "Microsoft Internet Explorer" bo iz tega seznama izvzet, ker ne podpira vseh ključnih funkcionalnosti za implementacijo knjižnice, in je bil zamenjan z naslednjo generacijo brskalnikov "Microsoft Edge"

Tabela 3.1: Verzije brskalnikov in delež uporabnikov (avgust 2017).

Mesto	**Verzija brskalnika**	**Delež trga**
1	Chrome for Android	27.18%
2	Chrome 60.0	16.68%
3	Safari iPhone	9.74%
4	Chrome 59.0	5.89%
5	IE 11.0	3.04%
6	Firefox 54.0	2.94%
7	Safari iPad	2.73%
8	UC Browser 11.3	2.47%
9	Android Browser	2.23%
10	Samsung Internet 5.4	1.96%
11	Safari 10.1	1.48%
12	Other	1.33%
13	UC Browser 10.9	1.32%
14	Firefox 55.0	1.19%
15	Chrome 49.0	1.12%
16	Edge 15	1.05%

Od izbranih brskalnikov skoraj vsi podpirajo EcmnaScript 6 funkcionalnosti [70]. Razen brskalnik safari na mobilnih napravah, ki imajo nameščen sistem iOS 9 ali manj, podpora pade pod 50%, vendar bomo skozi primere videli, ali so manjkajoče ključne funkcije.

3.3.1 Občutljive funkcionalnosti

Občutljive funkcionalnosti so določene na podlagi politik, ki jih predlaga ConScript, ter zlorabljene funkcionalnosti XSS napadov (poglavje 3.1).

Conscript politike [23] so:

- politika dinamičnega vstavljanja "script" značk,
- politika podajanja nizov v funkcije *setTimeout* in *setInterval*,
- politika vstavljanja kode v elemente HTML,
- politika filtriranja vir "script" značk,
- politika vstavljanja "noinline" značke,
- politika https sheme za klic zunanjih virov,
- politika dostopa do piškotkov,
- politika klicev meddomenskih virov,
- politika omejevanja klikanja povezav po dostopu do piškotkov,
- politika omejevanja pojavnih oken,
- politika vstavljanja okvirjev (iframe),
- politika filtriranje preusmeritev,
- politika omejevanja uporabe funkcij okolja,
- politika omejevanja uporabe selektorjev elementov HTML, ki so počasnejši,
- politika preprečitve vračanja praznega seznama po uporabi selektorjev,
- politika uporabe funkcije "eval" samo za serializacijo JSON zapisov.

Med navedenimi politikami manjkajo še omejitve prestrezanja dogodkov (ang. Events), preko katerih lahko koda sledi uporabnikovi interakciji s portalom. Prav tako bi pri drugi politiki (pretvorba niza v kodo) dodali še konstruktor funkcije, ki ima prav tako nevarno zmožnost pretvorbe funkcije v izvršljivo JavaScript funkcijo.

Za zagotavljanje CIA vidikov pri posameznih elementih HTML bi bilo treba vzpostaviti omejitve vstavljanja in dostopanja do posameznih elementov.

Vstavljanje značk HTML

Značke HTML lahko vstavimo v strukturo HTML preko več objektov in njihovih lastnosti. Pri iskanju funkcij pomaga prototipno drevo objektov JavaScript, saj na primer vsi elementi HTML dedujejo iz objekta Node, ki vsebuje lastnost *insertBefore*.

- **Node.prototype**
  - appendChild
  - insertBefore
  - replaceChild
  - **Element.prototype**
    - after
    - append
    - attachShadow
    - before
    - innerHTML
    - insertAdjacentElement
    - insertAdjacentHTML
    - outerHTML
    - prepend
    - replaceWith
  - **Document.prototype**
    - append
    - execCommand (možnost izvršitve ukazov, ki vstavijo HTML vsebino)
    - open
    - registerElement
    - write
    - writeln

Iskanje in premikanje po strukturi DOM

Objekti, ki predstavljajo strukturo DOM, imajo vgrajene funkcije, ki omogočajo prehode med elementi HTML.

- **Node.prototype**
  - childNodes
  - firstChild
  - getRootNode
  - lastChild
  - nextSibling
  - ownerDocument
  - parentElement
  - parentNode
  - previousSibling
  - **Element.prototype**
    - children
    - closest
    - firstElementChild
    - getElementsByClassName
    - getElementsByTagName
    - getElementsByTagNameNS
    - lastElementChild
    - nextElementSibling
    - previousElementSibling
    - querySelector
    - querySelectorAll
    - shadowRoot
    - webkitMatchesSelector
  - **HTMLDocument.prototype**
    - all
  - **Document.prototype**
    - body
    - children
    - documentElement
    - elementFromPoint
    - elementsFromPoint
    - firstElementChild
    - getElementById
    - getElementsByClassName
    - getElementsByName
    - getElementsByTagName
    - getElementsByTagNameNS
    - head
    - hidden
    - images
    - lastElementChild
    - links
    - querySelector
    - querySelectorAll
    - rootElement
    - scripts
    - scrollingElement

    - styleSheets
- Event.prototype
  - currentTarget
  - target
  - srcElement
- window
  - frames

Funkcije, ki lahko sprožijo zahtevek izven izvorne domene

Uhajanje podatkov preko HTTP zahtevkov je mogoče na tri načine. Prvi je uporaba namenskih funkcij za komuniciranje izven brskalnika (AJAX) [64]. Drugi način je zloraba nastavljanja "src" in "href" atributov na elementih z naslovom, ki vsebuje na primer piškotke [44]. Tretji način je zloraba preusmeritev in komunikacije med zavihki [8]. Funkcije, ki to omogočajo so:

- navigacijske funkcije:
  - window.location
  - document.location
  - location.push
  - location.href
  - location.replace
  - window.history.go
- klici zunanjih virov:
  - objekt XMLHttpRequest (url se nastavi preko metode *open*);
  - funkcija *fetch*
  - window.open
  - WebSocket
  - EventSource
  - HTMLImageElement.prototype.src
  - RTCPeerConnection

Atributi elementov HTML

- <a href=url >
- <applet codebase=url >
- <area href=url >
- <base href=url >
- <blockquote cite=url >
- <body background=url >

- <del cite=url >
- <form action=url >
- <frame longdesc=url >
- <frame src=url >
- <head profile=url >
- <iframe longdesc=url >
- <iframe src=url >
- <img longdesc=url >
- <img src=url >
- <img usemap=url >
- <input src=url >
- <input usemap=url >
- <ins cite=url >
- <link href=url >
- <object classid=url >
- <object codebase=url >
- <object data=url >
- <object usemap=url >
- <q cite=url >
- <script src=url >
- <audio src=url >
- <button formaction=url >
- <command icon=url >
- <embed src=url >
- <html manifest=url >
- <input formaction=url >
- <source src=url >
- <video poster=url >
- <video src=url >
- <img srcset="url1 resolution1 url2 resolution2" >
- <source srcset="url1 resolution1 url2 resolution2" >
- <object archive=url >ali <object archive="url1 url2 url3" >
- <applet archive=url >ali <applet archive=url1,url2,url3 >
- <meta http-equiv="refresh" content="seconds; url" >
-

Seznam atributov, ki vsebuje povezavo kot vrednost:

- href
- src
- background
- cite
- action
- longdesc
- poster
- formaction
- manifest
- data
- usemap
- srcset
- archive
- classid
- style (podniz, ki se začne pri "url")

Preostale pogoste JavaScript funkcije iz napadov XSS [44, 62].

- eval
- alert
- confirm
- location - navigacija okna
- document.cookie - piškotki
- navigator - dostop do sistemskih funkcij (geolokacija)

Dogodki in krmilniki dogodkov

Krmilniki dogodkov (ang. event handlers) so funkcije, ki se izvedejo ob sprožitvi dogodka na ustreznem objektu ali elementu HTML. V okolju JavaScript lahko krmilnike dogodkov nastavimo na dva načina. Prvi je preko namenskih funkcij na elementih HTML, kot so na primer "onclick". Drugi način

je z uporabo funkcij objekta *"EventTarget" - addEventListener('tipDogodka', krmilnik)* [4].

Namenske funkcije se lahko nahajajo na več objektih. Uporabniške krmilnike najdemo na objektih kot so "window" oz. globalni objekt, "Document", "Element" in "HTMLElement". Namenske lastnosti, ki sprejemajo krmilnike, se praviloma začnejo z nizom "on" ("ontipDogodka") [71].

3.3.2 Mehanizmi zaščite

Lahki samovarovani JavaScript je zaščitne funkcije vzpostavil po načelih aspektnega programiranja (slika 3.7) [11]. Po teh načelih "aspekt" vsebuje točko vstopa, ki definira točko vstavitve in pogoje zanjo ter nasvet (ang. advise), ki predpisuje potrebne spremembe [11].

```
var wrapper = function(objektMetoda, Politika){
  // Shranitev reference na originalno funkcijo
  var originalnaMetoda = objektMetoda.objekt[objektMetoda.metoda];

  // Nova funkcija
  var aspekt = function(){
    // Priprava končnih subjektov v lokalnem spominu
    var klic = {
      objekt : this,
      argumenti : arguments
    }

    //Klic politike s povratno funkcijo podano kot parameter
    return Politika.apply(klic.objekt, [klic.metoda, nadaljuj: function(){
      //Klic originalne metode v kontekstu orginalnega objekta
      originalnaMetoda.apply(klic.objekt, klic.argumenti)
    }])

  }
  //Zamenjeva metode
  objektMetoda.objekt[objektMetoda.metoda] = aspekt
  return aspekt
}
```

Slika 3.7: Primer vmesne funkcije po načelih aspektnega programiranja.

Vendar predlagani mehanizmi obsegajo bistveno večji del zmožnosti, kot jih lahki samovarovani JavaScript omogoča. Mehanizme lahko razdelimo v sledeče sklope:

- kontrola preusmeritev,
- sledenje spremembam strukture DOM,

- preprečitev vstavljanja elementov HTML,
- kontrola izvajanja ciljnih metod (eval itd.),
- filtriranje izvorov in zunanje komunikacije,
- kontrola dostopa do piškotkov,
- kontrola dostopa do podatkov naprave,
- preprečitev dostopa do izbranih elementov HTML.

Kontrola preusmeritev

Preusmeritve lahko izvedemo preko klicev funkcij v objektih za navigacijo in zgodovino [1]. Ker sta obe referenci na objekt za navigacijo "window.location" in "document.location" zaklenjeni za spreminjanje (configurable=false), ne moremo omejiti preusmeritev s klasičnimi vmesnimi funkcijami.

Vseeno obstaja možnost zaznave preusmeritve preko dogodkov "onbeforeunload" in "unload", vendar preko njiju ni mogoče preprečiti preusmeritve.

```
var location = window.document.location;

var prepreciNavigacijo = function () {
    var hash = location.hash;

    window.setTimeout(function () {
        location.hash = 'stop' + ~~ (9999 * Math.random());
        location.hash = hash;
    }, 0);
};

window.addEventListener('beforeunload', prepreciNavigacijo, false);
window.addEventListener('unload', prepreciNavigacijo, false);
```

Slika 3.8: Preusmeritev na originalno povezavo preden brskalnik prične nalaganje nove povezave.

Pristop, ki ga predstavlja koda na sliki 3.8, je sestavljen iz krmilnika, ki se sproži tik preden brskalnik prične preusmeritev. Ker spreminjanje naslova ne

[1] window.location, document.location, window.history

uspe znotraj krmilnika, je potrebno uporabiti trik s funkcijo "setTimeout". Slednja ima nastavljeno asinhrono funkcijo, ki se izvede takoj (zamik 0 ms). Brskalnik prične s preusmeritvijo na ciljno stran, vendar, ker takoj zatem pride do ponovne spremembe naslova, se preusmeritev prekliče in stran se vrne na prvotni naslov. Da je krmilnik skladen s portali, ki uporabljajo kriptografske izvlečke (ang. hash) pri navigaciji, se najprej izvede začasna preusmeritev na naključen naslov in potem hitra preusmeritev na izvorno stran.

Opisani pristop je zasnovan na predpostavki, da je izvajanje kode JavaScript hitrejše, kot je sposoben brskalnik pripraviti in izvesti zahtevek za prenos nove spletne strani. Z manjšimi dopolnitvami lahko dodamo preverjanje, ali nov naslov ustreza pogojem (slika 3.9). Ključne dopolnitve so prestavitev logike znotraj modula, dodajanje pogojev (ujemanje v začetnem delu naslova), ter ključavnici, ki kontrolirata zaporedje izvajanja, tudi kadar pride do spremembe naslov s silo (angl. brute force).

Za ustrezno zaščito krmilnika je potrebno poskrbeti, da se ne spreminjajo funkcije "setTimeout", "Array.prototype.filter" in "String.prototype.startsWith", da se izognemo težavam s spreminjanjem funkcij. Zato je potrebno kodo in kontrole izvesti pred kakršno koli drugo škodljivo vsebino. Priporočena je vstavitev značke 'script" z vsebino knjižnice v glavi dokumenta HTML, da se izvrši pred ostalimi skriptami [40].

Sledenje spremembam strukture DOM

Sledenje spremembam strukture DOM je v obstoječih pristopih temeljilo predvsem na takrat obstoječih funkcionalnostih. Ker se JavaScript in okolje nenehno dopolnjujeta [3], ni več mogoče uporabljati metode "watch", ki je omogočala prestrezanje sprememb objektov [11, 3]. Edini pristop, ki ostane, je uporaba novejšega konstrukta "MutationObserver", ki omogoča spremljanje sprememb elementov HTML. Na sliki 3.10 je primer sledenja spremembam, ki vstavljenim elementom odstrani atribut "onerror". Slednji se je uporabljal v XSS napadih za samodejno izvršitev funkcij na vstavljenih

```
(function () {
    "use strict"

    var location = window.document.location;

    // Kljucavnice
    var pass = {
        ok : false,
        redirecting : false
    }

    // Krmilnik
    var prepreciNavigacijo = function () {
        var hash = location.hash;

        // URL kamor preusmerja brskalnik
        var nextHref = location.href;

        // Ali je bila preusmeritev odobrena (prejsna prozitev krmilnika)
        if(!pass.ok){

            window.setTimeout(function () {
                //Hitro preusmeriu nazaj, da se izvedejo pogoji in
                // potem preusmeri na ciljno stran
                location.hash = '____' + ~~ (9999 * Math.random());
                location.hash = hash;

                // V postopku avtorizacije preusmerjanja
                pass.redirecting = true;

                //Preverjanje pogojev
                if(isURLWhitelisted(location.href)){

                    // Odobri preusmeritev
                    pass.ok = true;
                    // Preusmeritev
                    location.href = nextHref;

                }else{
                    // URL ni bil dovoljen
                    console.log(location.href)
                    throw 'URL not allowed'
                }
            }, 0);

        // Preusmeritev odobrena, ponastavi kljucavnice
        }else if(pass.redirecting){
            pass.ok = false;
            pass.redirecting = false;
        }
    }

    window.addEventListener('beforeunload', prepreciNavigacijo, false);
    window.addEventListener('unload', prepreciNavigacijo, false);
})();
```

Slika 3.9: Preverjanje preusmeritev z dopolnitvami.

slikah [44].

```
// izbrani HTML element
var target = document

// funkcija ob spremembi
var observer = new MutationObserver(function (mutations) {
    mutations.forEach(function (mutation) {

        mutation.addedNodes.forEach(function(node){
            if (node.hasAttribute && node.hasAttribute('onerror')) {
                node.removeAttribute('onerror')
            }
        })

    });
});

// konfiguracija sledenja:
var config = { attributes: true, childList: true, characterData: true, subtree: true }

// pričetek sledenja
observer.observe(target, config);
```

Slika 3.10: Sledenje spremembam strukture DOM in izvajanje ustrezni akcij.

MutationObserver omogoča, da pri posamezni spremembi lahko primerjamo stanje elementa pred in po spremembi. To nam omogoča, da na posameznih elementih vzpostavimo preverjanje in zagotavljanje integritete tako atributov kot tudi vsebine. Prav tako je tudi pri tem pristopu pomembno, da vse klice funkcij ustrezno zaščitimo pred spremembami (ang. tempering).

Preprečitev vstavljanja elementov HTML

MutationObserver omogoča tudi, da posamezne poskuse vstavitve elementov odpravimo (slika 3.11).

```
mutation.addedNodes.forEach(function(node){
    if (node.hasAttribute && node.hasAttribute('onerror')) {
        node.removeAttribute('onerror')
    }

    if(node.tagName === 'DIV'){
        node.remove()
    }

})
```

Slika 3.11: Z manjšo dopolnitvijo lahko preprečimo vstavitev "div" elementov.

Kontrola izvajanja ciljnih metod

Vmesne funkcije po principih aspektnega programiranja, kot je primer na sliki 3.7, je izhodiščna osnova za kontrolo izvajanja funkcij. Predlagana funkcija je dopolnitev vmesne funkcije, da je prilagojena za robustnejše delovanje. Ključne dopolnitve so lokalno shranjevanje referenc na zunanje funkcije in spremenjen del, ker se nastavlja nadomestna funkcija.

Na začetku funkcije opravimo nekaj osnovnih preverjanj, da zagotovimo ustrezno uporabo knjižnice. Reference na zunanje funkcije se shranijo lokalno v kontekstu funkcije.

JavaScript pozna tako podatkovne kot tudi dostopne lastnosti in jih je potrebno obravnavati ločeno. Dostopne lastnosti celo omogočajo ločeno kontroliranje pisanja in branja vrednosti (ang. getter and setter), zato v prvem delu nove funkcije (slika 3.12) podatkovne lastnosti pripravimo na pretvorbo v dostopno lastnost. Vrednost se shranjuje v interni skriti referenci in pripravita se dve enostavni funkciji za dostop in pisanje.

V drugem delu mehanizma (slika 3.13) se pripravita dve nadomestni funkciji za branje in pisanje. Parametri politike so tip dostopa do lastnosti (dostop in nastavljanje lastnosti), ime metode ter originalna funkcija, ki se kliče, če je

```
var secureFunction = (function(Object){
    // Lokalne reference
    var ObjectgetOwnPropertyDescriptor = Object.getOwnPropertyDescriptor
    var ObjectdefineProperty = Object.defineProperty

    var wrapper = function(objekt, metoda, Politika){
        if(objekt === undefined || metoda === undefined || !(Politika instanceof Function) ){
            throw 'Napačni vhodni tipi'
        }

        // Shranitev reference na originalno funkcijo
        var atributi = ObjectgetOwnPropertyDescriptor(objekt, metoda)

        if(atributi === undefined || atributi.configurable !== true){
            throw 'Lastnost ni konfigurabilna: '+metoda
        }

        var value = null; // Lokalna hramba dejanske vrednosti
        var getter = null, setter = null;

        if(atributi.value){
            // podatkovna lastnost
            // Hrani vrednost skrito
            value = atributi.value

            // nadomestne funkcije za dostopanje do vrednosti
            getter = function() { return value }
            setter = function (val) { value = val }

        } else {
            // dostopna lastnost
            // shrani reference na originalne funkcije
            getter = atributi.get
            setter = atributi.set
        }
```

Slika 3.12: Prvi del nove vmesne funkcije z dopolnitvami.

funkcija politike odločila, da je dostop odobren. Na koncu obe novi funkciji nastavimo namesto izvorne lastnosti ter preprečimo nadaljnje spreminjanje ali brisanje lastnosti.

Za primer lahko omenjeno varnostno funkcijo uporabimo za preprečitev klica metode "alert" (slika 3.14). Ob klicu ali nastavitvi funkcije se bo sprožila vmesna funkcija in izvedla politiko, ki klic razreši s sprožitvijo napake in ustavi izvajanje virtualnega stroja za dano funkcijo [3].

Filtriranje zunanje komunikacije in vstavljanje vsebin s tujim izvorom

Filtriranje izvorov se vzpostavi v dveh delih. Potrebno je kontrolirati vstavljanje elementov HTML, ki vsebujejo zunanji naslov v enem izmed navedenih atributov, in vzpostaviti nadzor nad konstrukti JavaScript, ki izvajajo zunanje klice (razdelek 3.1).

```
        // Nova ''beri'' funkcija
        var getAspekt = function(){
          //Klic politike s povratno funkcijo podano kot parameter
          // politika = function(tip, metoda, origFunkcija) - this = objekt
          return Politika.apply(this, ['GET', metoda, getter])
        }

        // Nova ''pisi'' funkcija
        var setAspekt = function(){
          //Klic politike s povratno funkcijo podano kot parameter
          // politika = function(tip, metoda, origFunkcija) - this = objekt
          return Politika.apply(this, ['SET', metoda, setter, arguments])
        }

        //Zamenjeva metode
        ObjectdefineProperty(objekt, metoda, {
            get : getAspekt,
            set : setAspekt,
            // prepreci brisanje ali prepisovanje nove lastnosti
            configurable : false
        })
    }

    // Skrivanje definicije funkcije
    return function(){
        return wrapper.apply(this, arguments)
    }

}(Object))
```

Slika 3.13: Drugi del nove vmesne funkcije z dopolnitvami.

```
// Primer restriktivne politike
secureFunction(window, 'alert', function(tip, metoda, cb){ throw 'Prepovedan klic'})
```

Slika 3.14: Primer uporabe nove funkcije.

Za filtriranje naslovov URL v elementih HTML uporabimo obstoječi sledilec sprememb strukture DOM. Dopolnimo ga s preverjanjem, ali vsebuje občutljive atribute in ali so le ti ustrezno nastavljeni. Primer take dopolnitev je na sliki 3.15. Prvi del preverja ustreznost atributa ob spremembah vrednosti na obstoječih elementih. Drugi del preverja atribute na elementih, ki so bili vstavljeni v strukturo DOM. Funkcija "checkAttibuteUrl" preveri, če se vrednost sklicuje na vir iz iste domene ali strani. To so naslovi, ki se začnejo z "#" (hash vrednosti - interna navigacija na strani [4, 3]) ter z "/" (lokalni viri, ki se jim doda na začetku ime gostitelja [4]). Paziti je le treba, da se naslov ne začne z dvema poševnicama, kar označuje vir iz določene domene z isto shemo kot trenutna stran [4].

Za zanesljivo delovanje funkcij je potrebno preprečiti spremembe funkcij:

- "String.prototype.startsWith",
- "Array.prototype.filter",
- "Array.prototype.map",
- "Array.prototype.forEach",
- "Array.prototype.filter",
- "Element.prototype.getAttributeNode",
- "NamedNodeMap.prototype.getNamedItem".

Zaščititi je potrebno tudi spremembe lastnosti na prototipih objektov Node, Element in Attr. V primeru da je katera koli izmed omenjenih funkcij spremenjena, se lahko mehanizem obide.

```
// funkcija ob spremembi
var urlSensitiveAttributes = ["href","src","background","cite","action","longdesc","poster"]
var okURLs = ['file:///home/mvehar'];
var blockedUrlHash = '#__BLOCKED'

var checkAttibuteUrl = function (attribute) {
    if(attribute.value === blockedUrlHash || attribute.value.startsWith('#')) return
    // isti izvor
    if(attribute.value.startsWith('/') && !attribute.value.startsWith('//')) return

    // ali se ujema vsaj z enim usreznim url
    var ok = okURLs.filter(prefix => attribute.value.startsWith(prefix)) > 0

    // ce je ujemanje prekini
    if(ok) return

    // Popravi attribut
    attribute.value = blockedUrlHash
}

var observer = new MutationObserver(function (mutations) {
    mutations.forEach(function (mutation) {

        if(mutation.type === 'attributes'){
            // Sprememba atributa na obstojecem elementu
            var changedAttribute = mutation.attributeName
            var attr = mutation.target.getAttributeNode(changedAttribute)

            if(attr !== null){
                checkAttibuteUrl(attr)
            }
        }

        mutation.addedNodes.forEach(function(node){
            // Vstavljeni elementi HTML
            if(node.attributes !== undefined){
                var attrs = node.attributes
                var attrs = urlSensitiveAttributes
                                .map((a) => attrs.getNamedItem(a))
                                .filter(a=> a!==null)

                attrs.forEach(a => checkAttibuteUrl(a))
            }

            if (node.hasAttribute && node.hasAttribute('onload')) {
                node.removeAttribute('onload')
            }

            if(node.tagName === 'DIV'){
                node.remove()
            }

        })

    });
});
```

Slika 3.15: Filtriranje naslovov v atributih elementov HTML.

Filtriranje naslovov pri programskih konstruktih, ki izvajajo zunanje klice, se izvede z redefinicijo le teh. Iz poglavja 3.1 je takih konstruktov malo. Konstruktorske funkcije lahko nadomestimo z vmesno politiko za konstruktorje, ki preverja ustreznost parametrov in nato izvede inicializacijo objekta (WebSocket, EventSource, RTCPeerConnection). Za funkcije, ki so del prototi-

pov, lahko uporabimo politiko, ki zamenja lastnosti prototipov ter pri klicu funkcije "window.open" uporabimo klasično zamenjavo lastnosti z vmesno funkcijo. Vsi primeri protiukrepov so prikazani na sliki 3.16.

```
var urlWhitelistPolicy = function(tip, metoda, origFunkcija){
    // onemogocimo prepis funkcij
    if(tip === 'SET') return undefined

    // ce so podani argumenti preverimo ali je URL ustrezen
    // URL je prvi argument
    var wrap = function(url){
        if(isURLWhitelisted(url)){
            // Izvedi funkcijo
            return origFunkcija().apply(this, arguments)
        } else {
            throw 'URL ni dovoljen'
        }
    }

    return wrap
}

var urlWhitelistSetterPolicy = function(tip, metoda, origFunkcija, args){
    // onemogocimo prepis funkcij
    if(tip === 'GET') return origFunkcija.call(this)

    if(tip === 'SET') {
        var url = args.length>0 ? args[0] : ''
        if(isURLWhitelisted(url)){
            return origFunkcija.apply(this, args)
        }else{
            throw 'URL ni dovoljen'
        }
    }
}

var urlWhitelistConstructorPolicy = function(tip, metoda, origFunkcija){
    // onemogocimo prepis funkcij
    if(tip === 'SET') return undefined

    // ce so podani argumenti preverimo ali je URL ustrezen
    // URL je prvi argument
    if(origFunkcija().prototype !== undefined){
       return function(url){
            if(isURLWhitelisted(url)){
                // Vrni instanco prototipa
                return Reflect.construct(origFunkcija(), arguments)
            } else {
                throw 'URL ni dovoljen'
            }
       }
    }
}
```

Slika 3.16: Politike za filtriranje naslovov URL v metodah, konstruktorskih funkcijah in metodah prototipov.

Politike za kontroliranje klicev AJAX so osnovne za prestrezanje metod. Edino pri metodi "XMLHttpRequest.prototype.open" je naslov URL na dru-

gem mestu med parametri metode.

```
secureFunction(XMLHttpRequest.prototype, 'open', urlWhitelistPolicy)
secureFunction(HTMLImageElement.prototype, 'src', urlWhitelistSetterPolicy)

secureFunction(window, 'fetch', urlWhitelistPolicy)
secureFunction(window, 'open', urlWhitelistPolicy)

secureFunction(window, 'WebSocket', urlWhitelistConstructorPolicy)
secureFunction(window, 'EventSource', urlWhitelistConstructorPolicy)
secureFunction(window, 'RTCPeerConnection', urlWhitelistConstructorPolicy)
```

Slika 3.17: Aktiviranje politik za filtriranje naslovov URL.

Za zanesljivo delovanje politik je treba preprečiti spremembe metod "Reflect.contruct", "Function.prototype.apply", "Function.prototype.call".

Kontrola dostopa do piškotkov

Kontrolo dostopa do piškotkov lahko vzpostavimo na preprost način z osnovno politiko za metode, kot je prikazano na slike 3.19. Funkcija za nastavljanje in dostopanje do piškotkov se nahaja na prototipu objekta "Document". V tem primeru smo omejili možnost branja piškotkov, dovolimo samo dodajanje novih piškotkov.

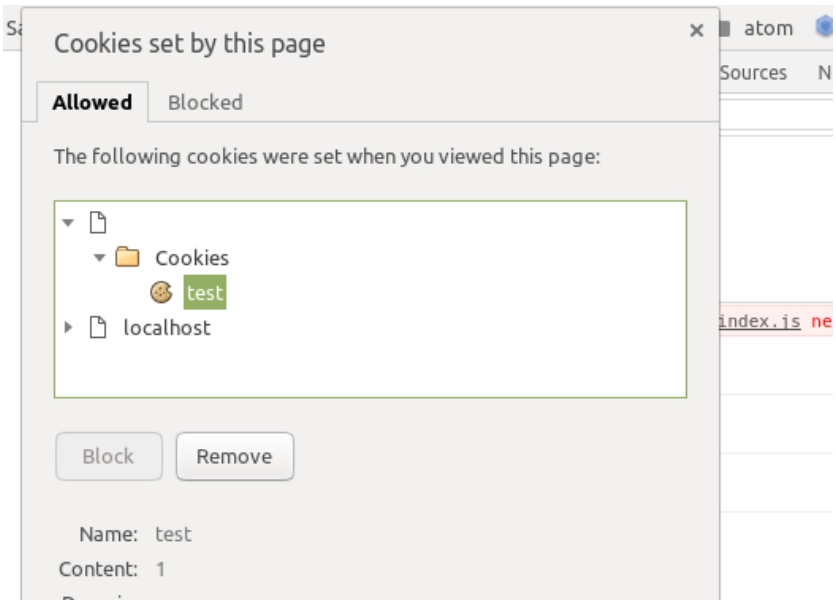
```
var writeOnlyPolicy = function(tip, metoda, origFunkcija, args){
    // onemogocimo prepis funkcij
    if(tip === 'GET') return undefined

    return origFunkcija.apply(this, args)

}
```

Slika 3.18: Politika, ki dovoljuje samo nastavljanje lastnosti.

```
secureFunction(Document.prototype, 'cookie', writeOnlyPolicy)
```

Slika 3.19: Aktiviranje politike za piškotke.

Slika 3.20: Testiranje mehanizma za piškotke.



Slika 3.21: Dokaz delovanja politike za piškotke.

```
var noAccessPolicy = function(){
    // onemogocimo prepis funkcij
    throw 'Dostop zavrnjen'
}

secureFunction(window, 'navigator', noAccessPolicy)
```

Slika 3.22: Politika, ki absolutno zavrne dostop do funkcije.

Kontrola dostopa do podatkov naprave

Podatke in funkcije za podatke o napravi lahko pridobimo preko lastnosti "window.navigator" [40]. V kolikor naša spletna stran ne uporablja takšnih funkcionalnosti, kot so geolokacija ali podatki o brskalniku, lahko dostop do njih preventivno onemogočimo z enostavno politiko, ki zavrača vsak poskus dostopa.

Preprečitev dostopa do izbranih elementov HTML

Kontrola dostopa do posameznih elementov HTML zahteva obsežnejši pristop. Do elementov lahko dostopamo preko iskalnih funkcij, kot na primer "document.getElementById", ali pa preko funkcij za sprehajanje po strukturi DOM. Dostop do posameznih elementov lahko dobimo tudi preko dogodkov v krmilnikih dogodkov. Opis funkcij, ki so namenjene opisanim namenom, najdemo v poglavju 3.1. Vse te funkcije so del prototipov, zato jih bomo nadomestili s politikami za prototipne metode.

Pri drugem delu mehanizma je potrebno določiti osnovna pravila, do katerih elementov lahko dostopamo. Za prikaz delovanja bo izbran način določitev elementov s posebno vrednostjo atributa "class". Glede na to vrednost bo na začetku mogoče elemente izbrati iz strukture DOM in jim nastaviti skrite lastnosti, ki bodo odločale, ali je element skrit ali ne.

Vse navedene funkcije, ki vračajo elemente, bomo spremenili tako, da bomo pred vračanjem elementov izločili tiste, ki so označeni kot skriti.

S pomočjo štirih politik lahko kontroliramo dostop do lastnosti, ki vračajo enega ali več elementov HTML, ter do funkcij, ki omogočajo iskanje elementov.

```
var HIDE_ELEMENTS_WITH_CLASS = "____HIDEME____";

const filterElements = function(elements){
    if(elements === null) return elements
    return Array.from(elements).filter(el => el.HIDEME !== true)
}

const filterElement = function(element){
    if(element === null) return element
    return element.HIDEME === true ? null : element
}

const htmlMultipleOutHideFnProlicy = function(tip, metoda, origFunkcija, args){
    // onemogocimo prepis funkcij
    if(tip === 'SET') return undefined

    // ce so podani argumenti preverimo ali je URL ustrezen
    // URL je prvi argument
    var wrap = function(){
        var out = origFunkcija().apply(this, arguments)

        return filterElements(out)
    }
    return wrap
}

const htmlMultipleOutHideProlicy = function(tip, metoda, origFunkcija, args){
    // onemogocimo prepis funkcij
    if(tip === 'SET') return undefined

    // ce so podani argumenti preverimo ali je URL ustrezen
    // URL je prvi argument
    var out = origFunkcija.apply(this, arguments)

    return filterElements(out)
}

const htmlSingleOutHideFnProlicy = function(tip, metoda, origFunkcija, args){
    // onemogocimo prepis funkcij
    if(tip === 'SET') return undefined

    // ce so podani argumenti preverimo ali je URL ustrezen
    // URL je prvi argument
    var wrap = function(){
        var out = origFunkcija().apply(this, arguments)

        return filterElement(out)
    }
    return wrap
}

const htmlSingleOutHideProlicy = function(tip, metoda, origFunkcija, args){
    // onemogocimo prepis funkcij
    if(tip === 'SET') return undefined

    // ce so podani argumenti preverimo ali je URL ustrezen
    // URL je prvi argument
    var out = origFunkcija.apply(this, arguments)

    return filterElement(out)
}
```

Slika 3.23: Politike za preverjanje ali je skriti element v rezultatu funkcij oziroma lastnosti.

```
// mutiple out - functions
secureFunction(Document.prototype, 'getElementsByClassName', htmlMultipleOutHideFnProlicy)
secureFunction(Document.prototype, 'getElementsByTagName', htmlMultipleOutHideFnProlicy)
secureFunction(Document.prototype, 'elementsFromPoint', htmlMultipleOutHideFnProlicy)

secureFunction(Element.prototype, 'getElementsByClassName', htmlMultipleOutHideFnProlicy)
secureFunction(Element.prototype, 'getElementsByTagName', htmlMultipleOutHideFnProlicy)
secureFunction(Element.prototype, 'getElementsByTagNameNS', htmlMultipleOutHideFnProlicy)
secureFunction(Element.prototype, 'querySelectorAll', htmlMultipleOutHideFnProlicy)


// multiple out
secureFunction(Node.prototype, 'childNodes', htmlMultipleOutHideProlicy)
secureFunction(Element.prototype, 'children', htmlMultipleOutHideProlicy)
secureFunction(Document.prototype, 'children', htmlMultipleOutHideProlicy)
secureFunction(Document.prototype, 'hidden', htmlMultipleOutHideProlicy)
secureFunction(Document.prototype, 'images', htmlMultipleOutHideProlicy)
secureFunction(window, 'frames', htmlMultipleOutHideProlicy)
secureFunction(Document.prototype, 'links', htmlMultipleOutHideProlicy)
secureFunction(Document.prototype, 'scripts', htmlMultipleOutHideProlicy)


// single out
secureFunction(Node.prototype, 'parentNode', htmlSingleOutHideProlicy)
secureFunction(Node.prototype, 'firstChild', htmlSingleOutHideProlicy)
secureFunction(Node.prototype, 'getRootNode', htmlSingleOutHideProlicy)
secureFunction(Node.prototype, 'lastChild', htmlSingleOutHideProlicy)
secureFunction(Node.prototype, 'ownerDocument', htmlSingleOutHideProlicy)
secureFunction(Node.prototype, 'parentElement', htmlSingleOutHideProlicy)
secureFunction(Node.prototype, 'previousSibling', htmlSingleOutHideProlicy)

secureFunction(Element.prototype, 'closest', htmlSingleOutHideProlicy)
secureFunction(Element.prototype, 'firstElementChild', htmlSingleOutHideProlicy)
secureFunction(Element.prototype, 'lastElementChild', htmlSingleOutHideProlicy)
secureFunction(Element.prototype, 'nextElementSibling', htmlSingleOutHideProlicy)
secureFunction(Element.prototype, 'previousElementSibling', htmlSingleOutHideProlicy)

secureFunction(Document.prototype, 'body', htmlSingleOutHideProlicy)
secureFunction(Document.prototype, 'head', htmlSingleOutHideProlicy)
secureFunction(Document.prototype, 'documentElement', htmlSingleOutHideProlicy)
secureFunction(Document.prototype, 'rootElement', htmlSingleOutHideProlicy)
secureFunction(Document.prototype, 'lastElementChild', htmlSingleOutHideProlicy)
secureFunction(Document.prototype, 'firstElementChild', htmlSingleOutHideProlicy)

// single out - functions
secureFunction(Element.prototype, 'querySelector', htmlSingleOutHideFnProlicy)
secureFunction(Element.prototype, 'webkitMatchesSelector', htmlSingleOutHideFnProlicy)
secureFunction(Document.prototype, 'elementFromPoint', htmlSingleOutHideFnProlicy)
secureFunction(Document.prototype, 'scrollingElement', htmlSingleOutHideFnProlicy)


//Event
secureFunction(Event.prototype, 'currentTarget', htmlSingleOutHideProlicy)
secureFunction(Event.prototype, 'target', htmlSingleOutHideProlicy)
secureFunction(Event.prototype, 'srcElement', htmlSingleOutHideProlicy)
```

Slika 3.24: Impliciranje politik za celosten sistem varovanja dostopa do skritih elementov HTML.

Struktura DOM se spreminja, med izvajanjem dodamo v sledilnik sprememb dodatno preverjanje, če se pojavi element z nastavljenim atributom.

```
if(node.classList && node.classList.contains(HIDE_ELEMENTS_WITH_CLASS)){
    Object.defineProperty(node, 'HIDEME', {
        value : true,
        configurable : false
    })
}
```

Slika 3.25: Nastavljanje skritih elementov ob spremembah strukture DOM.

Preprečitev spreminjanja izbranih elementov HTML

MutationObserver omogoča ob spremembah pridobitev prejšnje vrednosti. S hitrim popravkom spremembe lahko dosežemo obstojnost podatkov v elementu HTML, kot je prikazano na sliki 3.26.

Večje vprašanje se postavlja, kaj storiti ob brisanju celotnega drevesa elementov znotraj strukture DOM. Možnosti so preprečitev brisanja preko *MutationObserver*-ja ali pa vzpostaviti kontrole kot za dostop do skritih elementov HTML.

Preprečitev nastavitev krmilnikov dogodkov

Nastavitev krmilnikov dogodkov je mogoča preko atributov elementov HTML (npr. onclick) ali preko priporočenih metod [64] addEventListener('click', function krmilnik()...) [3].

Kontrola atributov je mogoča preko sledenja spremembam elementov HTML z *MutationObserver* ter z onemogočanjem lastnosti na prototipih objektov elementov HTML. Slednje se enostavno izvede s politiko za preprečitev uporabe lastnosti (slika 3.22).

Kontrola uporabe metode "addEventListener" se vstavi na prototipnem objektu *EventTarget*. S politiko, ki nadzira uporabo funkcije lahko preprečimo nastavitev poljubnih dogodkovnih krmilnikov na izbranih objektih, ki razširjajo prototip *EventTarget*.

```
const NOCHANGES_CLASS = '____INTEGRITY_____'
const integrityFunction = function (mutations) {
    mutations.forEach(function (mutation) {

        if(mutation.type === 'attributes'){
            // Sprememba atributa na obstojecem elementu
            var changedAttribute = mutation.attributeName
            var attr = mutation.target.getAttributeNode(changedAttribute)

            attr.value = mutation.oldValue
            return
        }

        if(mutation.type === 'characterData'){
            // Sprememba atributa na obstojecem elementu
            mutation.target.data = mutation.oldValue
        }
    }
}

const elementIntegrity = function(element, options){

    var observer = new MutationObserver(integrityFunction);
    observer.observe(element, options)
}

(function(){
    const elements = document.getElementsByClassName(NOCHANGES_CLASS)
    const options = {
        attributes: true, characterData: true,
        attributeOldValue: true, characterDataOldValue : true
    }

    if(elements){
        Array.from(elements).forEach(el => elementIntegrity(element, options))
    }
})()
```

Slika 3.26: Funkcije za zagotavljanje integritete izbranega elementa.

3.4 Uporaba knjižnice

Knjižnica bo ustrezno delovala samo, če se ustrezni ukrepi za zaščito vzpostavijo, kar se da hitro po začetku procesiranja dokumenta HTML. Dokument HTML se razčlenjuje znak za znakom v strukturo DOM. Takoj, ko je zaključena značka "script", se pretvarjanje ustavi in izvrši se koda, ki jo vsebuje značka HTML [4]. Če vključimo našo knjižnico na sam začetek glave dokumenta (ang. head), se politike in nastavitve izvršijo pred ostalo kodo v neonesnaženem okolju JavaScript. Če spletna stran vsebuje dopolnilno kodo (ang. polyfills), ki po možnosti neinvazivno spreminja definicije osnovnih metod, moramo knjižnico umestiti v izvajanje po njej. Tako bodo politike dobile reference do ustreznih funkcij, ki bodo zamenjale vmesne metode. Novih funkcij ni mogoče nadomestiti, ker so zaščitene na ravni navideneznega

stroja JavaScript.

Politike lahko aktiviramo kadarkoli, ko presodimo, da ne bo več potreb po klicih lastnosti v nasprotju s politikami. Dodatno si lahko pred aktivacijo politike lokalno shranimo referenco do originalnih funkcij, vendar je v tem primeru ogrožena učinkovitost kontrol, saj lahko reference zelo hitro uidejo izven izoliranih kontekstov. Priporočena je uporaba JavaScript modulov, ki za preproste sekvenčne operacije zagotavlja izolacijo od globalnega okolja.

Pri vsakem delu mehanizma je tudi navedeno, da je potrebno zagotoviti, da ne pride do sprememb ključnih metod, ki jih funkcije uporabljajo. Ta težava je pestila obstoječe metode zaščite skozi objektne zmožnosti jezika JavaScript [68]. Težave se lahko reši z zamrznitvijo objektov (poglavje 3.2.2) ali pa posameznih metod z nastavitvijo atributa "configurable" na "false".

Pri navedbi brskalnikov, za katere je bila knjižnica pripravljena, je bilo izpostavljeno, da mobilni brskalnik Safari ne podpira obsežnejšega dela specifikacije ECMAScript 6. Vendar skozi pregled funkcionalnosti, ki so bile uporabljene za pripravo knjižnice, ugotovimo, da so ključne funkcije za zamenjavo lastnosti in sledenje sprememb strukture DOM na voljo. Potrebno je le paziti pri deklaraciji spremenljivk z določilom "let" in uporabo puščičnih funkcij. Oboje se lahko reši z uporabo določila "var" ter uporabo navadnih deklaracij funkcij.

Predstavljene politike v tem delu so samo testni primeri preprostih politik, ki lahko nadzorujejo izvajanje občutljivih konstruktov. Vsaka izmed politik se tudi osredotoča na določen pristop k varovanju, naj si bo to sledenje spremembam strukture DOM ali kontroliranje izvajanja določenih metod. Vse te politike se lahko poljubno prilagodi in uporablja tudi za potrebne zaščite drugih konstruktov.

3.5 Vpliv na delovanje

Samo knjižnico bomo testirali glede vpliva pri nalaganju strani in glede vpliva na izvajanje osnovnih funkcij. Primerjava bo izvedena zgolj za primere, ko so pogoji uspešno izpolnjeni, saj pri neizpolnjenih pogojih pride do takojšnje ustavitve izvajanja funkcije.

Testi bodo izvedeni na gostitelju z 32GB pomnilnika, Intel i5 6700Q 8-jedrnim procesorjem (4,3Ghz na jedro), 250 GB NVME pomnilnikom (Samsung 750) z nameščenim operacijskim sistemom MATE Linux 16.04. Testi bodo izvedeni v brskalnikih Chrome (verzija 61) ter Firefox (verzija 55), ki predstavljata zadnji verziji dveh izmed najpogostejših brskalnikov.

Zaščitni mehanizmi se funkcionalno delijo glede na to, ali se ciljne metode nadomesti s kontroliranimi ali se nadomesti konstrukcijska funkcija, ter posamezni pristopi k kontroliranju sprememb strukture DOM. S spodaj naštetimi testi tako pokrijemo vse različne pristope, ki jih knjižnica in politike uporabljajo.

Testi funkcionalnosti knjižnice:

- dodajanje ustreznih značk (div);
- spreminjanje zaščitenih značk (celovitost);
- klicanje kontrolirane metode;
- kreiranje instance objekta WebSocket z ustreznim naslovom URL;
- klicanje metod za izbiranje elementov HTML;
- klicanje metod za seznam pod elementov HTML.

Testi bodo izvedeni večkrat po več iteracij funkcij pred in po aktiviranju politik.

3.5.1 Test 1 - dodajanje ustreznih značk

Test 1 prikazuje razliko v času izvajanja metode vstavitve elementa HTML v strukturo DOM. Vstavljali smo značko img z ustreznim atributom "src". Testi so bili izvedeni desetkrat po deset tisoč iteracij za izvorno in kontrolirano verzijo okolja.

Testi so pokazali, da je upočasnitev delovanja funkcij opazna samo v brskalniku Google Chrome - 40,55%. V brskalniku Mozilla Firefox so rezultati primerljivi z izvirno funkcijo.

3.5.2 Test 2 - spreminjanje zaščitenih značk

Test 2 prikazuje vpliv zaščite celovitosti izbranih značk na hitrost izvedbe funkcij za vstavitev elementov HTML. Testi so bili izvedeni desetkrat po deset tisoč iteracij za izvorno in kontrolirano verzijo okolja.

Testi so pokazali, da je upočasnitev delovanja funkcij opazna samo v brskalniku Google Chrome - 79,14%. V brskalniku Mozilla Firefox so rezultati primerljivi.

3.5.3 Test 3 - klicanje kontroliranih metod

Test 3 prikazuje vpliv politik na izvajanje izvirnih metod. Test je bil izveden s permisivno politiko, ki ne izvaja nobenih preverjanj, temveč neposredno izvede ciljno funkcijo.

Testi so pokazali, da je upočasnitev delovanja funkcij opazna samo v brskalniku Google Chrome - 67,38%. V brskalniku Mozilla Firefox so rezultati primerljivi z izvirno funkcijo.

3.5.4 Test 4 - kreiranje instance objekta WebSocket z ustreznim naslovom URL

Test 4 preverja vpliv na delovanje konstruktorskih funkcij. Za test bo uporabljen konstrukt WebSocket in naslov testnega strežnika `wss://echo.websocket.`

`org`. Izvedenih je bilo deset serij po petdeset tisoč klicev izvirnih in nato spremenjenih konstruktov.

Testi so pokazali, da je upočasnitev delovanja funkcij opazna v obeh brskalnikih. V brskalniku Google Chrome - 15,42%, v Mozilli Firefox pa počasnejši za 34,97%.

3.5.5 Test 5 - klicanje metod za iskanje elementov HTML

Test 5 prikazuje vpliv kontrol za skrivanje elementov HTML na hitrost izvajanja funkcij. Uporabljeni sta bili metodi "document.getElementById" in "document.getElementsByTagName". V obeh metodah se je preverjalo, ali izhodni podatki vsebujejo skrite elemente. Testi so bili izvedeni v desetih serijah po dvajset tisoč klicev.

Rezultati so pokazali upočasnitev na brskalniku Chrome za 921,43% ter v brskalniku Firefox za 242,11%.

3.5.6 Test 6 - klicanje metod za seznam pod elementi HTML

Zadnji test je preverjal vpliv kontrol za dostop do podseznama elementov HTML v posameznih elementih HTML. Deset serij po dvajset tisoč klicev funkcije "children.length" na testnem elementu je pokazalo drastično upočasnitev v obeh brskalnikih. V Google Chromu za 32550% ter v Mozilli Firefox za 14850%. Razlogi za tako drastično upočasnitev so v optimizacijah pogonov JavaScript, ki imajo število elementov shranjeno, medtem ko vmesna politika izvede pregled vseh elementov v izhodu originalne funkcije.

Tabela 3.2: Mozilla Firefox 55 - čas izvajanja testov v milisekundah.

# test	original	st. dev.	kontrol.	st. dev.	upočasnitev
1	772 ms	18 ms	766 ms	13,65 ms	99,22%
2	650,44 ms	11,39 ms	675,8 ms	21,69 ms	103,90%
3	1601,8 ms	20,92 ms	1588,9 ms	56,68 ms	99,19%
4	248,2 ms	2,77 ms	335 ms	5,01 ms	134,97%
5	3,8 ms	0,45 ms	13 ms	4,47 ms	342,11%
6	0,8 ms	0,84 ms	119,6 ms	3,7 ms	14950,00%

Tabela 3.3: Google Chrome 61 - čas izvajanja testov v milisekundah.

# test	original	st.dev.	kontrol.	st. dev.	upočasnitev
1	110 ms	42,97 ms	154,6 ms	13,76 ms	140,55%
2	74,3 ms	30,33 ms	133,1 ms	6,67 ms	179,14%
3	356,8 ms	18,46 ms	597,2 ms	24,38 ms	167,38%
4	517,4 ms	18,46 ms	597,2 ms	27,36 ms	115,42%
5	2,8 ms	2,05 ms	31,4 ms	6,54 ms	1121,43%
6	1,2 ms	0,84 ms	391,8 ms	3,9 ms	32650,00%

Pri testiranju se je izkazalo, da daljše izvajanje posameznih delov kode sproži mehanizme brskalnika proti nedelovanju spletne strani, kar je vplivalo na rezultate testov. Zaradi tega so bili testi izvedeni v manjših sklopih naenkrat in v več serijah.

Testi so pokazali, da je upočasnitev zaznavna samo pri mehanizmih, ki kontrolirajo interakcijo s strukturo DOM, ker imajo brskalniki vzpostavljene optimizacije za sprehajanje po seznamih elementov (angl. lazy evaluation). Ker knjižnica preveri vsak element HTML, ali je dosegljiv se dejansko izvede dostop do vsakega elementa v seznamu ter nato vrne seznam.

3.6 Razširitve knjižnice

Knjižnica prikazuje zmožnosti jezika JavaScript za vzpostavitev kontrol okolja, v katerem sobivajo vsebine. Obvarovanje primarne vsebine se mora vzpostaviti takoj na začetku dokumenta HTML in samega izvajanja, da ne pride do onesnaženja okoljskih konstruktov in metod. Zaradi uporabe spletnih tehnologij za vzpostavitev kontrol je mogoče knjižnico uporabiti tudi kot dodatek brskalniku. Tak dodatek bi lahko omogočil uporabo določenih politik, ki bi onemogočale uhajanje občutljivih informacij. Obenem lahko razvijalci knjižnico in politike uporabijo za preverjanje uporabe funkcionalnosti s strani tujih knjižnic ter jim dinamično dovolijo uporabo.

V razvoju pa so tudi določeni pristopi, ki bi jih lahko razvijalci uporabili za okrepitev varnosti izvajanja lastne kode in primarne vsebine. To so predvsem razvoj senčne strukture DOM in konstrukti Realm.

Senčna struktura DOM je eksperimentalna funkcionalnost brskalnikov, ki omogočajo, da v del strukture DOM vstavimo elemente HTML, ki so vidni uporabnikom, vendar so izven dosega kode in oblikovnih določil primarnega okolja. Po intuiciji so podobni značkam iframe z razliko, da koda znotraj senčne strukture lahko dostopa do ostalih delov strani.

Konstrukt JavaScript Realm je predlog za dopolnitev pogona JavaScript z objekti, ki omogočajo izolirano izvajanje kode z ločenim globalnim objektom brez možnosti dostopa do strukture DOM. Prav tako tudi ta pristop spominja na značko iframe in tudi trenutna dopolnilna koda se zanaša za prenos izvajanja v konstrukte iframe ter konstrukte Workers.

Poglavje 4

Zaključki in ugotovitve

Obstoječi pristopi k varovanju vsebine so učinkovit način zaščite pred grožnjami, vendar se je razvoj spletnih tehnologij dodobra spremenil. JavaScript ni več samo dodatek k vsebini HTML za poživitev portalov, ampak je postal osrednji del predstavitve informacij na odjemalcih. Temu pričajo predvsem pospešen razvoj novih funkcionalnosti, priljubljenost pri uporabi knjižnic in ogrodij, kot so Angular in React, ki postavljata JavaScript v ospredje delovanja spletne strani.

Po drugi strani je uporaba načela zmožnosti (ang. capabilities) pri zagotavljanju varnosti platform prisotna od začetkov razvoja operacijskih sistemov [72]. Preprostost in učinkovitost zmožnostnih sistemov se je soočala s težavami v implementaciji, saj vsak dober zmožnosten sistem močno temelji na karakteristikah podpornih sistemov [9]. JavaScript kot pomnilniško varen jezik predstavlja dobro izhodišče za njegovo implementacijo, vendar brez poenotenja med brskalniki ni bilo mogoče pričakovati uporabnih rešitev [40].

Predlagana knjižnica je nadaljevanje obstoječih idej, ki so jih imeli avtorji rešitve Caja po uporabi internih zmožnosti okolja in programskega jezika [10, 28]. Obseg je bil dodatno raziskovalno razširjen na celosten pristop skozi vidike CIA. Tako je nastalo ogrodje za aktiviranje politik, ki vsebuje mehanizme za omejevanje spletnih napadov (XSS). Predstavljeni so bili primeri, kako lahko posamezne funkcije - zmožnosti okolja - omejimo z vmesnimi funk-

cijami po načelu spletnega programiranja [11]. Napake preteklih rešitev [68] so izpostavile pomanjkljivosti takih pristopov, zato je knjižnica zasnova za optimalno varno delovanje in varovalne ukrepe, ki napadalcem onemogočajo onesposobitev.

Testiranje knjižnice na sodobnih brskalnikih je pokazalo sicer upočasnitev določenih funkcij (neposredno testiranje potrebnega časa za izvedbo funkcije), vendar se v praktični uporabi in ob asinhroni naravi jezika spletna vsebina izvaja brez opaznih zamikov. Te ugotovitve tudi sovpadajo z obstoječim pristopom z lahkimi samovarovanimi vmesnimi funkcijami [11], kjer so se posamezne operacije izvajale tudi do desetkrat počasneje, vendar ni bilo opaznega makro vpliva na delovanje spletne vsebine.

Razvoj spletnih tehnologij je v polnem zagonu. EcmaScript 6 je bila izdana leta 2015 [3] in jo do danes novejši brskalniki dodobra podpirajo. Pripravlja se nova različica in spletne aplikacije dobivajo vrsto novih zmožnosti preko vmesnikov na brskalniku [4]. Razvijajo se tudi drugačni pristopi k ureditvi spletnih vsebin, kot je na primer senčena struktura DOM [27] in spletne komponente [73]. Spletne tehnologije niso več samo spletne (Node.js, Electron itd.), temveč so postale najbolj uspešen primer izvajanja mobilne kode v tem trenutku.

Koda, politike in testi so dosegljivi na spletnem mestu GitHub (`https://github.com/mvehar/JavaScriptSecurityEnchantments`).

Literatura

[1] N. Bielova, Survey on JavaScript Security Policies and their Enforcement Mechanisms in a Web Browser , Journal of Logic and Algebraic Programming 82 (Št. 8) (2013) 243–262.

[2] Usage of javascript for websites, Dostopno na: `https://w3techs.com/technologies/details/cp-javascript/all/all`, (14.9.2017).

[3] E. Standard, 262: Ecmascript language specification, 6th edition, december 1999, Dostopno na: `https://www.ecma-international.org/ecma-262/6.0/`(1.9.2017).

[4] S. Faulkner, A. Danilo, A. Eicholz, T. Leithead, HTML 5.1 2nd edition, Candidate recommendation, W3C, https://www.w3.org/TR/2017/CR-html51-20170620/ (1.9.2017) (2017).

[5] How i got xss'd by my ad network, Dostopno na: `https://www.troyhunt.com/how-i-got-xssd-by-my-ad-network/`, (1.9.2017).

[6] S. Ali, S. Khusro, A. Rauf, A cryptography-based approach to web mashup security (2011) 53–57.

[7] M. Ter Louw, J. S. Lim, V. N. Venkatakrishnan, Enhancing web browser security against malware extensions, Journal in Computer Virology 4 (Št. 3) (2008) 179–195.

[8] J. Grossman, XSS Attacks: Cross-site scripting exploits and defense, Syngress, (2007).

[9] M. S. Miller, K.-P. Yee, J. Shapiro, et al., Capability myths demolished, Tech. rep., Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www. erights. org/elib/capability/duals (2003).

[10] S. Maffeis, J. C. Mitchell, A. Taly, Object capabilities and isolation of untrusted web applications, 2010 IEEE Symposium on Security and Privacy (2010) 125–140.

[11] P. H. Phung, D. Sands, A. Chudnov, Lightweight self-protecting javascript, Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (2009) 47–60.

[12] M. E. Whitman, H. J. Mattord, Principles of information security, Cengage Learning, (2011).

[13] T. Berners-Lee, WWW: past, present, and future, IEEE Computer 29 (10) (1996) 69–77.

[14] M. Vandenwauver, J. Claessens, W. Moreau, C. Vaduva, R. Maier, Why enterprises need more than firewalls and intrusion detection systems, Enabling Technologies: Infrastructure for Collaborative Enterprises, 1999.(WET ICE'99) Proceedings. IEEE 8th International Workshops on (1999) 152–157.

[15] B. Stone-Gross, M. Cova, C. Kruegel, G. Vigna, Peering through the iframe, 2011 Proceedings IEEE INFOCOM (2011) 411–415.

[16] B. Stritter, F. Freiling, H. König, R. Rietz, S. Ullrich, A. von Gernler, F. Erlacher, F. Dressler, Cleaning up web 2.0's security mess-at least partly, IEEE Security Privacy 14 (Št. 2) (2016) 48–57.

[17] X. Xiao, R. Yan, R. Ye, Q. Li, S. Peng, Y. Jiang, Detection and prevention of code injection attacks on html5-based apps, 2015 Third International Conference on Advanced Cloud and Big Data (2015) 254–261.

[18] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, G. Vigna, Cross site scripting prevention with dynamic data tainting and static analysis., Network and Distributed System Security Symposium, (2007).

[19] Wanglu, Majun, Chenjing, Zhangxindong, Design of code security detection system based on rule inspection, 2010 International Conference on Information, Networking and Automation (ICINA) 2 (2010) 346–349.

[20] G. C. Necula, P. Lee, Safe, untrusted agents using proof-carrying code, Mobile Agents and Security (1998) 61–91.

[21] J. Li, D. Yu, L. Maurer, A resource management approach to web browser security, International Conference on Computing, Networking and Communications (2012) 697–701.

[22] S. Stamm, B. Sterne, G. Markham, Reining in the web with content security policy, Proceedings of the 19th international conference on World wide web (2010) 921–930.

[23] L. A. Meyerovich, B. Livshits, Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser, IEEE Symposium on Security and Privacy (2010) 481–496.

[24] D. Yu, A. Chander, N. Islam, I. Serikov, Javascript instrumentation for browser security, ACM SIGPLAN Notices 42 (Št. 1) (2007) 237–249.

[25] R. N. Shah, K. R. Patil, Securing third-party web resources using subresource integrity automation, IJETT 4 (Št. 2) (2017).

[26] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, S. Esmeir, Browsershield: Vulnerability-driven filtering of dynamic html, ACM Transactions on the Web (TWEB) 1 (Št. 3) (2007) 61–74.

[27] P. D. Ryck, N. Nikiforakis, L. Desmet, F. Piessens, W. Joosen, Protected web components: Hiding sensitive information in the shadows, IT Professional 17 (Št. 1) (2015) 36–43.

[28] S. Van Acker, A. Sabelfeld, Javascript sandboxing: Isolating and restricting client-side javascript (2016) 32–86.

[29] A. Grosskurth, M. W. Godfrey, A case study in architectural analysis: The evolution of the modern web browser. emse (2007).

[30] E. Shepherd, Same origin policy for javascript, Na naslovu: `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`(1.9.2017).

[31] M. Smith, Http access control (cors), Na naslovu: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS`(1.9.2017).

[32] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, G. Vigna, You are what you include: large-scale evaluation of remote javascript inclusions, Proceedings of the 2012 ACM conference on Computer and communications security (2012) 736–747.

[33] D. B. Lange, M. Oshima, Seven good reasons for mobile agents, Communications of the ACM 42 (Št. 3) (1999) 88–89.

[34] R. D. Dean, Formal aspects of mobile code security, Ph.D. thesis, Princeton, NJ, USA (1999).

[35] D. Kotz, R. S. Gray, Mobile agents and the future of the internet, SIGOPS Oper. Syst. Rev. 33 (Št. 3) (1999) 7–13.

[36] O. Tripp, P. Ferrara, M. Pistoia, Hybrid security analysis of web javascript code via dynamic partial evaluation, Proceedings of the 2014 International Symposium on Software Testing and Analysis (2014) 49–59.

[37] M. Hefeeda, B. Bhargava, On mobile code security, Center of education and Research in Information Assurance and Security And Department of Computer Science, Purdue University West Lafayette, IN, (2002).

[38] Content security policy level 3, Dostopno na: `https://www.w3.org/TR/CSP3/`, (1.9.2017).

[39] W. Jansen, Countermeasures for mobile agent security, Computer Communications 23 (Št. 17).

[40] C. McCormack, Web idl, World Wide Web Consortium(1.9.2017).

[41] L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne, Document object model (dom) level 3 core specification (2004).

[42] S. Jiang, S. Smith, K. Minami, Securing web servers against insider attack, Computer Security Applications Conference, 2001. ACSAC 2001. Proceedings 17th Annual.

[43] Owasp top 10 application security risks - 2017, Dostopno na: `https://www.owasp.org/index.php/Top_10_2017-Top_10`, (1.9.2017).

[44] Html5 security cheatsheet, Dostopno na: `http://html5sec.org`, (1.9.2017).

[45] W. Jansen, T. Karygiannis, Mobile agent security, Tech. rep., NATIONAL INST OF STANDARDS AND TECHNOLOGY GAITHERSBURG MD (1998).

[46] S. Stamm, B. Sterne, G. Markham, Reining in the web with content security policy, Proceedings of the 19th international conference on World wide web (2010) 921–930.

[47] W. W. W. Consortium, et al., HTML 4.01 Specification, WWW Consortium, 1999.

[48] W3schools try it: Framesets, Dostopno na: `https://www.w3schools.com/tags/tryit.asp?filename=tryhtml_frame_cols`, (1.9.2017).

[49] P. Jagnere, Vulnerabilities in social networking sites, Parallel Distributed and Grid Computing (PDGC), 2012 2nd IEEE International Conference on (2012) 463–468.

[50] N. Provos, P. Mavrommatis, M. A. Rajab, F. Monrose, All your iframes point to us, Proceedings of the 17th Conference on Security Symposium (2008) 1–15.
URL http://dl.acm.org/citation.cfm?id=1496711.1496712

[51] Whatwg, html 5 specification, Dostopno na: https://html.spec.whatwg.org/, (1.9.2017).

[52] A. Nitze, Evaluation of javascript quality issues and solutions for enterprise application development, International Conference on Software Quality (2015) 108–119.

[53] K.-I. D. Kyriakou, I. K. Chaniotis, N. D. Tselikas, The gpm meta-transcompiler: Harmonizing javascript-oriented web development with the upcoming ecmascript 6 "harmony" specification, 2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC) (2015) 176–181.

[54] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, F. Piessens, Jsand: complete client-side sandboxing of third-party javascript without browser modifications, Proceedings of the 28th Annual Computer Security Applications Conference (2012) 1–10.

[55] H. M. Levy, Capability-based computer systems, Digital Press, (2014).

[56] M. S. Miller, Robust composition: Towards a unified approach to access control and concurrency control, Ph.D. thesis, Johns Hopkins University, Baltimore, Maryland, USA (May 2006).

[57] S. Maffeis, A. Taly, Language-based isolation of untrusted javascript, Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE.

[58] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, W. Joosen, Webjail: least-privilege integration of third-party components in web mashups, Proceedings of the 27th Annual Computer Security Applications Conference (2011) 307–316.

[59] L. Ingram, M. Walfish, Treehouse: Javascript sandboxes to help web developers help themselves., USENIX Annual Technical Conference (2012) 153–164.

[60] D. Crockford, Adsafe: Making javascript safe for advertising (2008).

[61] M. S. Miller, et al., Secure ecmascript 5, Na naslovu: `https://code.google.com/archive/p/es-lab/wikis/SecureEcmaScript.wiki`.

[62] Owasp cross-site scripting (xss), Dostopno na: `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`, (1.9.2017).

[63] C. Heilmann, Unobtrusive javascript, Lulu Enterprises Incorporated, (2005).

[64] C. Yue, H. Wang, Characterizing insecure javascript practices on the web, Proceedings of the 18th international conference on World wide web (2009) 961–970.

[65] Ransomware that's 100% pure javascript, no download required, Dostopno na: `https://nakedsecurity.sophos.com/2016/06/20/ransomware-thats-100-pure-javascript-no-download-required/`, (1.9.2017).

[66] G. Richards, S. Lebresne, B. Burg, J. Vitek, An analysis of the dynamic behavior of javascript programs, ACM Sigplan Notices 45 (Št. 6) (2010) 1–12.

[67] Web apis, Dostopno na: `https://developer.mozilla.org/en-US/docs/Web/API`, (1.9.2017).

[68] J. Magazinius, P. H. Phung, D. Sands, Safe wrappers and sane policies for self protecting javascript., NordSec 7127 (2010) 239–255.

[69] Browser version market share worldwide, Dostopno na: `http://gs.statcounter.com/browser-version-market-share`, (1.9.2017).

[70] Browser compatibility with es6, Dostopno na: http://kangax.github.io/compat-table/es6/, (1.9.2017).

[71] Global event handlers mixin, Dostopno na: https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers, (1.9.2017).

[72] R. Y. Kain, C. E. Landwehr, On access checking in capability-based systems, IEEE Transactions on Software Engineering (Št. 2) (1987) 202–207.

[73] Web components, Dostopno na: https://developer.mozilla.org/en-US/docs/Web/Web_Components, (1.9.2017).