SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 232

Tehnike otkrivanja zloćudnog kôda na web stranicama

Jakov Zubčić

Zagreb, srpanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 12. ožujka 2021.

ZAVRŠNI ZADATAK br. 232

Pristupnik:	**Jakov Zubčić (0036517598)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	doc. dr. sc. Stjepan Groš

Zadatak: **Tehnike otkrivanja zloćudnog kôda na web stranicama**

Opis zadatka:

Na Internetu je dostupno mnoštvo kompromitiranih i lažnih web sjedišta koja sadrže zloćudni kôd. Lažna web sjedišta mogu primjerice nuditi preuzimanje zaraženih dokumenata, a kompromitirane web trgovine mogu sadržavati zloćudni JavaScript kôd koji unesene osobne podatke i podatke o kreditnim karticama prosljeđuje napadačima. U sklopu završnoga rada potrebno je istražiti tehnike kojima se s klijentske strane može otkriti zloćudni kôd na web stranicama, razviti rješenje koje na temelju preuzetih web stranica i popratnih sadržaja nastoji pronaći i identificirati zloćudni kôd, te provesti evaluaciju razvijenog rješenja nad skupom podataka i prokomentirati dobivene rezultate. Rješenje mora uključivati testove i biti robusno. Radu priložiti izvorni kôd razvijenih i korištenih programa. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 11. lipnja 2021.

Zahvaljujem se doc. dr. sc Stjepanu Grošu i mag. ing. Ivanu Kovačeviću na pruženoj pomoći i savjetima pri izradi ovog rada, te svojoj obitelji i prijateljima za podršku koju su mi pružali tijekom studiranja.

SADRŽAJ

1. Uvod

Tema ovog rada je zloćudni kod na web stranicama, te tehnike njegove detekcije. Internet ima ogroman broj korisnika, u 2019. je taj broj bio 3,97 milijardi [12] i na njemu je ogromna količina osobnih podataka, čijom krađom napadači mogu imati financijske dobiti. Zbog toga je Internet vrlo pogodna platforma za širenje zloćudnog koda. Najkorišteniji programski jezik u današnje vrijeme u razvoju klijentske strane web stranica je JavaScript, koristi ga preko 97% web stranica [8] i zbog toga je zloćudni kod pisan u JavaScriptu izuzetno aktualna tema. Zloćudni kod može imati razne funkcionalnosti, neke od čestih su takozvani *Drive-By Download* [4], gdje se na korisnikovo računalo preuzimaju maliciozne datoteke bez da je korisnik svjestan toga, lažna web sjedišta mogu prosljeđivati osobne podatke napadačima i slično. Sam pojam zloćudnog koda, te njegove podjele i detaljnija objašnjenja će biti razrađena u poglavlju 2.1.

Cilj ovog rada je razviti programsko rješenje koje će nastojati uspješno otkriti zloćudni kod, te dati pregled postojećih tehnika za detekciju zloćudnog koda.

U razvijenom programskom rješenju za detekciju malicioznog koda korištena su YARA pravila. Ona su alat pomoću kojeg se može definirati skup uzoraka, njihove kombinacije i uvjete koji moraju biti ispunjeni kako bi se datoteka prepoznala kao maliciozna. Također je korišten antivirusni softver otvorenog koda ClamAV. Ove dvije tehnologije će također biti opisane u poglavljima 2.3 i 2.4.

Područje sigurnosti se neprestano razvija; napadači su svjesni metoda koje se primjenjuju u svrhu obrane od napada, pa aktivno razvijaju načine kako zaobići tu obranu. U slučaju zloćudnog koda pokušava se izbjeći prepoznavanje zloćudnosti koda od raznih antivirusnih programa. Postupak koji napadači u tu svrhu često koriste je obfuskacija koda.

Obfuskacija koda je postupak kojim se izvorni kod mijenja tako da postane nerazumljiv tako da funkcionalnost koda bude teško prepoznatljiva kako ljudima tako i programskim rješenjima koja žele odrediti je li kod zloćudan. Obfuskacija će biti detaljnije opisana u poglavlju 3.

Najopćenitija podjela analize zloćudnog koda je na dinamičku i statičku analizu.

Dinamička analiza podrazumijeva izvršavanje izvornog koda u izoliranoj okolini (engl. *sandbox*) kako bi se promatranjem ponašanja koda donijelo zaključak o tome je li kod zloćudan ili nije. Statička analiza se bavi analizom izvornog koda bez pokretanja u istu svrhu. U poglavlju 4 će biti objašnjeni neki pristupi problema detekcije zloćudnog koda korištenjem statičke i dinamičke analize.

U poglavlju 5 bit će opisano implementirano rješenje, bit će dan pregled pristupa koji su se nakon eksperimentiranja pokazali neuspješni, te će se analizirati rezultati implementiranog rješenja.

2. Osnovni pojmovi i korištene tehnologije

2.1. Zloćudni kod

Zloćudni kod (engl. *malware*) je programski kod u kojem je namjerno uključena funkcionalnost kojoj je cilj da se izvrši štetna namjera [29]. Ova definicija ne određuje konkretne tehnologije koje su korištene za ostvarivanje štetne namjere, platformu za koju je zloćudni kod napisan, niti njegove načine pokretanja i širenja. Iz toga je vidljivo da je zloćudan kod jedan vrlo širok pojam, zbog toga se često koriste razni izrazi koji detaljnije opisuju zloćudni kod temeljeno na raznim podjelama po karakteristikama [29]. U nastavku će biti predstavljene neke česte vrste zloćudnih kodova.

Vjerojatno najčešće spominjana vrsta zloćudnog koda su virusi. Glavna karakteristika virusa je to što se šire ubacivanjem u izvršne kodove legitimnih programa i pokreću se pokretanjem tih legitimnih programa. Ti legitimni programi nazivaju se "virusovim domaćinima". Virusi se dakle, oslanjaju na to da će njihovi domaćini u kojima su ubačeni biti pokrenuti kako bi izvršili svoju zloćudnu funkcionalnost.

Crvi su vrsta zloćudnog koda specifična po svom načinu širenja. Naime, za razliku od virusa, crvima nisu potrebni domaćini kako bi se širili. Crvi se šire po mreži izvršavajući svoj kod neovisno o drugim programima. Traže ranjive servise, dijeljene diskove i slične načine na koji se mogu proširiti. Obično im je cilj proširiti se na što veći broj računala.

Trojance (engl. *Trojan horses*) karakterizira predstavljanje da imaju nekakvu korisnu funkciju, dok zapravo izvodi neku neautoriziranu malicioznu funkciju, poput slanja pritisaka tipki na tipkovnici (engl. *key logging*) napadačima.

Iznuđivački zloćudni kod (engl. *Ransomware*) je zloćudni kod koji iznuđuje vlasnika zaraženog računala. To se najčešće radi tako da se sadržaj diska računala žrtve kriptira kriptografskom funkcijom, pa se zatim traži uplata novčanog iznosa na račun napadača, koji zauzvrat obećavaju poslati kriptografski ključ kako bi žrtva mogla de-

kriptirati podatke te im ponovno mogla pristupiti. Iznuđivački zloćudni kod je vrlo čest jer napadačima nije kompliciran za napisati, a donosi im relativno veliku zaradu.

Špijunski zloćudni kod (engl. *Spyware*) potajno prosljeđuje napadačima važne žrtvine podatke poput lozinki iz njihovih web preglednika i slično.

Downloaderi su vrsta zloćudnog koda kojoj je svrha preuzeti i instalirati na žrtvino računalo neki drugi zloćudni kod.

Programski jezik JavaScript je napadačima pogodan za pisanje zloćudnog koda jer pri pristupanju nekoj web stranici web preglednik automatski izvršava JavaScript kod. Primjer iskorištavanja te činjenice su tzv. *Drive-By Download* napadi. Kod tih napada se učitavanjem neke web stranice pokreću alati za iskorištavanje ranjivosti (engl. *exploit kits*) koji traže ranjivosti u web pregledniku i instaliranim dodacima, pa ih iskorištavaju kako bi preuzeli maliciozni kod na žrtvino računalo. Ovakvi napadi su vrlo potentni zbog toga što žrtva nije svjesna da je zloćudni kod preuzet te zbog toga što nije potrebna bilo kakva interakcija žrtve kako bi se napad realizirao - dovoljan je samo pristup web stranici.

Zloćudni kod na Internetu može je prisutan na raznim lažnim zloćudnim web stranicama, ali također i na legitimnim web sjedištima koja su kompromitirana. Na lažne zloćudne web stranice se često pokušava navesti žrtve da im pristupe *phishingom*. *Phishing* je postupak društvenog inženjeringa kojim se pokušava žrtvu na prevaru natjerati da pristupe zloćudnoj web stranici ili otkriju važne osobne podatke.

Zloćudan kod može dospijeti na legitimna web sjedišta *malvertisingom* [10] i iskorištavanjem *Cross-Site Scripting* (XSS) ranjivosti [15] ili drugih ranjivosti na web sjedištu. Postupkom *malvertisinga* napadači mogu kompromitirati web sjedišta iznajmljivanjem reklamnog prostora. Kod tog postupka, ako web stranica nema adekvatnu zaštitu, napadači mogu zakupiti reklamni prostor i koristeći tu reklamu pohraniti zloćudni kod na stranici. Drugi način kako napadači mogu ubaciti zloćudni kod na legitimne web stranice, XSS, je ranjivost na web stranicama kod koje se uneseni podaci na web stranicu tumače kao JavaScript kod koji se u nekom trenutku izvršava. Napadači tako mogu na stranici koja nema zaštitu protiv XSS-a ubaciti svoj kod koji će onda biti pohranjen na web stranici i pokretan kad joj korisnici budu pristupali. Zloćudni kod može dospjeti na legitimne web stranice i ako napadači iskorištavanjem neke ranjivosti uspiju dobiti pristup administracijskom dijelu stranice u slučajevima kada se koristi Wordpress [14] ili neki sličan *Content Management System* (CMS), ili kontrolu nad web poslužiteljem. U tim slučajevima napadači lako mogu dodati zloćudni kod na web stranicu.

2.2. Zloćudni kod u JavaScriptu

Zloćudni kod u programskom jeziku JavaScript se može podijeliti po tome koje ranjivosti iskorištava.

Zloćudni kod može iskorištavati ranjivosti u web pregledniku, ranjivosti Adobe Flasha, Adobe PDF readera, te mnogih dodataka (engl. *plugins*) web preglednika. Postoje i alati za iskorištavanje ranjivosti, koji su rastući trend na crnom tržištu i zaslužni su za čak 61% primijećenih zloćudnih aktivnosti na Internetu [25]. Neki poznatiji primjeri takvih alata su Angler [1], WebAttacker [13] i Nuclear [13].

U JavaScriptu postoje mnoge funkcije koje predstavljaju sigurnosni rizik jer se mogu vrlo lako zloupotrijebiti. Primjerice funkcija *eval()* može se koristiti za dinamičko generiranje koda koji se potom izvršava, *document.write()* se može koristiti za umetanje zloćudnog koda, te razne funkcije nad stringovima se mogu koristiti u svrhu obfuskacije zloćudnog koda kako bi se izbjegla detekcija. U tablici 2.1 dan je pregled nekih takvih funkcija, njihov opis i sigurnosni rizik kojeg predstavlja njihovo korištenje.

Tablica 2.1: Tablica nekih JavaScript funkcija sa sigurnosnim rizicima [25]

Ime funkcije	**Vrsta funkcije**	**Moguće prijetnje**
eval() window.setInterval() window.setTimeout()	Dinamičko izvršavanje koda	Dinamičko generiranje koda
location.replace() location.assign()	Promjena trenutnog URL-a	Preusmjeravanje na zlćudni URL
getUserAgent() getAppName()	Provjera korištenog web preglednika	Iskorištavanje ranjivosti određenog web preglednika
getCookie() setCookie()	Pristup kolačićima	Manipulacija kolačićima
document.addEventListener() element.addEventListener()	Presretanje događaja	Blokiranje korisnikovih radnji ili njihovo simuliranje
document.write() element.changeAttribute() document.writeln() element.innerHTML() element.insertBefore() element.replaceChild() element.appendChild()	Operacije na DOM-u HTML-a	Umetanje zlćudnih skripti
String.charAt() String.charCodeAt() String.fromCharCode() String.indexOf() String.split()	Operacije nad stringovima	Sakrivanje namjere korištenjem za obfuskaciju

Još jedna važna karakteristika JavaScripta jest to da on nije preveden (engl. *compiled*) jezik, nego interpretiran. To otežava njegovu analizu u smislu da ne postoje izvršne datoteke koje bi se moglo analizirati u svrhu detekcije zlćudnosti, nego je potrebno isključivo na temelju izvornog koda ili izvršavanja donijeti odluku o tome je li zlćudan.

Kada bi JavaScript imao i izvršni kod osim izvornog, mogla bi se na obje vrste koda primjeniti YARA pravila. YARA pravila su učinkovita u prepoznavanju ključnih

funkcionalnosti zloćudnog koda u izvršnom i izvornom formatu. Međutim, u slučaju programskog jezika JavaScript, su primjenjiva samo na izvorni kod. YARA pravila su opisana detaljnije u potpoglavlju 2.3.

2.3. YARA pravila

U razvijenom programskom rješenju korištena su YARA pravila [16]. Ovo poglavlje će opisati što su ona i kako se koriste. YARA (*Yet Another Recursive Acronym*) pravila su alat koji služi za identifikaciju i klasifikaciju zloćudnog koda. Pomoću YARA pravila se mogu definirati uzorci u tekstu ili binarnom zapisu koji su svojstveni za neku skupinu ili pojedini primjerak zloćudnog koda. Svako YARA pravilo ima svoje ime i sastoji se od 3 glavna dijela.

Prvi dio YARA pravila jesu metapodaci. Metapodatke čine razne informacije o pravilu, poput autora pravila, vremena stvaranja pravila, datuma modificiranja pravila i slično. Ovaj dio ne mora biti prisutan kako bi YARA pravilo bilo ispravno definirano, ali u praksi vrlo često je. Navođenjem metapodataka olakšava se snalaženje u pravilima, pogotovo onima koji ih nisu napisali, a žele ih koristiti.

Drugi dio YARA pravila su stringovi. Tu se definiraju stringovi koji će biti traženi u datoteci koja se ispituje. Podržani su stringovi u heksadecimalnom zapisu, tekstualni stringovi i regularni izrazi. YARA pravila također imaju podršku za korištenje *wildcardova*, definiranje raspona i sličnih opcija.

Treći dio YARA pravila su uvjeti. Uvjeti su logički izrazi koji moraju vrijediti kako bi se datoteka klasificirala kao zloćudna. U uvjetima se kao varijable koriste stringovi definirani u prethodnom dijelu, te se na njih primjenjuju razni operatori kako bismo od njih dobili logički izraz. Uvjeti podržavaju standardne logičke operatore disjunkcije, konjukcije i negacije, te razne relacijske operatore poput operatora veće od, jednako, manje od i slično.

U isječku 2.1 je dan jednostavan primjer YARA pravila koje će se podudarati s izvornim kodom koji koristeći naredbu *alert()*, naredbu *console.log()* ili obje, ispisuje riječ "primjer". Pravilo započinje ključnom riječi *rule* i njegovo ime je "YARA_primjer". U metapodacima su navedeni opis, ime autora i datum stvaranja pravila. U pravilu su definirana dva stringa koja će se tražiti u analiziranoj datoteci, *$s1_console* i *$s1_alert*. Nazivi stringova moraju započinjati znakom "$". U posljednjem dijelu je naveden uvjet koji govori da u analiziranoj datoteci mora biti pronađen barem jedan od dva definirana stringa kako bi se analizirana datoteka podudarala s pravilom. Pravilo će se, dakle, podudarati s izvornim kodom koji u sebi sadrži barem jednu od naredbi "*alert('primjer')*"

i "*console.log('primjer')*".

Isječak 2.1: Primjer jednostavnog YARA pravila

```
rule YARA_primjer {
   meta:
      description = "Primjer YARA pravila za završni rad"
      author = "Jakov Zubčić"
      date = "25-05-2021"
   strings:
      $s1_console = "alert('primjer')'"
      $s2_alert = "console.log('primjer')"
   condition:
      $s1_console or $s2_alert
}
```

2.4. ClamAV

ClamAV je besplatan antivirusni softver otvorenog koda (engl. *open-source*) dostupan za razne operacijske sustave [3]. Može se koristiti iz naredbenog retka (engl. *command-line*) ili kao višedretveni poslužiteljski proces (engl. *daemon*). U bazi podataka ClamAV-a je sadržano preko 5 milijuna primjeraka zloćudnog koda koje može prepoznati i ta baza podataka se redovito ažurira novim primjercima. Kako bi se mogao koristiti iz programskog jezika Python, potrebno je instalirati *clamd* poslužiteljski proces, pokretanjem naredbi u naredbenom retku prikazanih u isječku 2.2.

Isječak 2.2: Instalacija i pokretanje ClamAV poslužiteljskog procesa na operacijskom sustavu Ubuntu

```
$ sudo apt-get install clamav-daemon clamav-freshclam
  clamav-unofficial-sigs
$ sudo freshclam
$ sudo service clamav-daemon start
```

Potrebno je i pokrenuti naredbu u isječku 2.3 kako bi se instalirala Python biblioteka clamd.

Isječak 2.3: Instalacija Python biblioteke clamd

```
$ pip install clamd
```

ClamAV se može koristiti iz naredbenog retka pozivanjem naredbe clamscan. U isječku 2.5 je dan ispis programa za pokrenutu naredbu u isječku 2.4. Zastavica "-i" označava ispisivanje isključivo zaraženih datoteka, a zastavica "-r" označava rekurzivno posjećivanje direktorija.

Isječak 2.4: Pokretanje ClamAV-a iz naredbenog retka

```
$ clamscan -r -i .
```

Isječak 2.5: Ispis ClamAV-a u naredbenom retku

```
...
20160915_53c5edf2043c336282f7636088487d07.js:
Sanesecurity.Malware.26329.JsHeur.UNOFFICIAL FOUND
20160111_c24477753cd3b461496367e3d7d12a23.js:
Sanesecurity.Malware.25965.JsHeur.UNOFFICIAL FOUND
20160621_3369ae7054154ef4fbc05fe18488e1cd.js:
Win.Trojan.Locky-30618 FOUND
...
----------- SCAN SUMMARY -----------
Known viruses: 8658865
Engine version: 0.103.2
Scanned directories: 234
Scanned files: 38250
Infected files: 32174
Data scanned: 5302.47 MB
Data read: 4145.69 MB (ratio 1.28:1)
Time: 303.545 sec (5 m 3 s)
Start Date: 2021:05:28 16:25:55
End Date:   2021:05:28 16:30:59
```

U ispisu je vidljivo da je od ukupno 38250 zloćudnih JavaScript i HTML datoteka, preuzetih s [7], detektirano njih 32174. Također je dana informacija o kojoj vrsti zloćudnog koda je riječ u svakoj od pronađenih zloćudnih datoteka.

3. Obfuskacija izvornog koda

U ovom poglavlju će biti detaljnije objašnjen postupak obfuskacije izvornog koda. Obfuskacija je metoda kojom se mijenja struktura izvornog koda kako bi izvorni kod postao nerazumljiv i ljudima i softveru koji treba prepoznati je li kod zloćudan. Obfuskacijom se ne mijenja semantika koda, to jest obfuskacijom se ne mijenja funkcionalnost koda. Često se koristi kod zloćudnog koda kako bi se otežala njegova detekcija.

S obzirom na to da je izvorni kod klijentske strane dostupan svima koji pristupaju web stranici, obfuskacija se također može koristiti u benigne svrhe. Neki od razloga obfuskacije benignog koda mogu biti da se sakrije važna poslovna logika ili spriječi plagiranje. U nastavku će biti prikazani neki česti načini obfusciranja koda, opisani u [27]. Tehnike obfusciranja koda koje će biti opisane u ovom poglavlju su obfuskacija poslučajenjem, obfuskacija podataka, obfuskacija enkodiranjem i obfuskacija logičke strukture.

3.1. Obfuskacija poslučajenjem

Obfuskacija poslučajenjem je najjednostavnija tehnika obfuskacije. Kod ovog postupka dijelovi koda se nasumično umeću i mijenjaju se neki elementi koda. S obzirom na to da JavaScript interpreter ignorira znakove praznine i komentare, oni se bez mijenjanja semantike koda mogu umetati na slučajnim mjestima u kodu, što smanjuje čitljivost izvornog koda. Mijenjanje elemenata koda obuhvaća imena funkcija i varijabli; dodjeljuju im se nasumično stvorena, besmislena imena, ponovno s ciljem otežavanja razumijevanja koda. Obfuskacija poslučajenjem je demonstrirana na primjeru u isječku 3.1. Prvi dio primjera je izvorni kod prije obfuskacije poslučajenjem, a drugi dio nakon.

Isječak 3.1: Primjer obfuskacije poslučajenjem

```
//---------------------------------------------------------
// Početni isječak izvornog koda:
//---------------------------------------------------------
```

```
function ispisiString(str) {
    document.write(str);
}
let nekiString = "Ovo je primjer obfuskacije poslučajenjem.";
ispisiString(nekiString);
//-------------------------------------------------------------
// Isječak koda nakon obfuskacije poslučajenjem:
//-------------------------------------------------------------
function                w1H3ScAP3z(kAyFnBTyPa) {document.
write//gTBOvsQ2JN7Il73N1PzhgTBOvsQ2JN7Il73N1Pzh
(//0zVIDJPCYwCngItWhiz19pw3YWdIsb4a6Lf7CeQV
        kAyFnBTyPa//vsWn5WanTGe3sDIEgrbO
);}
let         oMiil18jcb =  /*oyXITyI471oyXITyI471*/
    "Ovo je primjer obfuskacije poslučajenjem."
w1H3ScAP3z( oMiil18jcb/*iRp1dnT2lyiRp1dnT2ly*/);
```

3.2. Obfuskacija podataka

Kod postupka obfuskacije podataka vrijednost varijabli ili konstanti postaje rezultat operacija nad jednom ili više varijabli ili konstanti. Najčešći oblici ove tehnike obfuskacije su razdvajanje stringova i zamjena ključnih riječi.

Razdvajanje stringova se radi tako da se jedan string zapiše kao rezultat konkatenacije nekoliko manjih stringova. Tako zapisani stringovi se znaju onda koristiti kao argumenti funkcija *document.write()* ili *eval()* koje izvršavaju taj string kao naredbu u pregledniku. Zamjena ključnih riječi je postupak kod kojeg se ključne riječi programskog jezika JavaScript zapisuju u varijable, pa se te varijable koriste umjesto ključnih riječi. Primjer ovog postupka obfuskacije dan je u isječku 3.2.

Isječak 3.2: Primjer obfuskacije podataka

```
//-------------------------------------------------------------
// Početni isječak izvornog koda:
//-------------------------------------------------------------

console.log("Primjer obfuskacije podataka");

//-------------------------------------------------------------
// Isječak koda nakon provedenog razdvajanja stringova:
//-------------------------------------------------------------
```

```
let ra = "cons";
let zdv = "ole.";
let aj = "lo"
let anj = "g(\""
let e_ = "Primjer"
let st = " obf"
let r = "uskaci"
let ing = "je pod"
let ov = "ataka\""
let a = ")"
eval(ra + zdv + aj + anj + e_ + st + r + ing + ov + a)
// izvršava se console.log("Primjer obfuskacije podataka")

//------------------------------------------------------------
// Isječak koda nakon provedene zamjene ključne riječi:
//------------------------------------------------------------

var zamjena = console;
zamjena.log("Primjer obfuskacije podataka");
```

3.3. Obfuskacija enkodiranjem

Kod postupka obfuskacije enkodiranjem izvorni kod se prikazuje *escapeanim* ASCII znakovima, u Unicode-u ili heksadecimalnim prikazom. Postoji i pristup gdje napadač koristi vlastitu funkciju za kodiranje znakova pa uz nju priloži i funkciju za dekodiranje koja onda dekodira kodirani kod prilikom izvođenja. Primjer ove obfuskacije je prikazan u isječku koda 3.3.

Isječak 3.3: Primjer obfuskacije heksadecimalnom zapisu

```
//------------------------------------------------------------
// Početni isječak izvornog koda:
//------------------------------------------------------------

console.log("primjer")

//------------------------------------------------------------
// Isječak koda u heksadecimalnom zapisu:
//------------------------------------------------------------

console.log("\x70\x72\x69\x6d\x6a\x65\x72")
// ponovno se ispisuje "primjer"
```

3.4. Obfuskacija logičke strukture

Obfuskacija logičke strukture je mijenjanje dijelova u kodu koji utječu na tijek izvođenja programa, ali tako da semantika ostane ista. Ovo se može postići na nekoliko načina.

Jedan način je umetanje "mrtvog koda" (*Dead Code Injection*). Taj postupak nastoji obfuscirati kod umetanjem naredbi grananja čiji uvjet nikada neće biti istinit, pa se posljedično neće nikada izvršiti.

Drugi način jest umetanje dodatnih logičkih struktura koji povećavaju kompleksnost ali ne mijenjaju funkcionalnost originalnog izvornog koda.

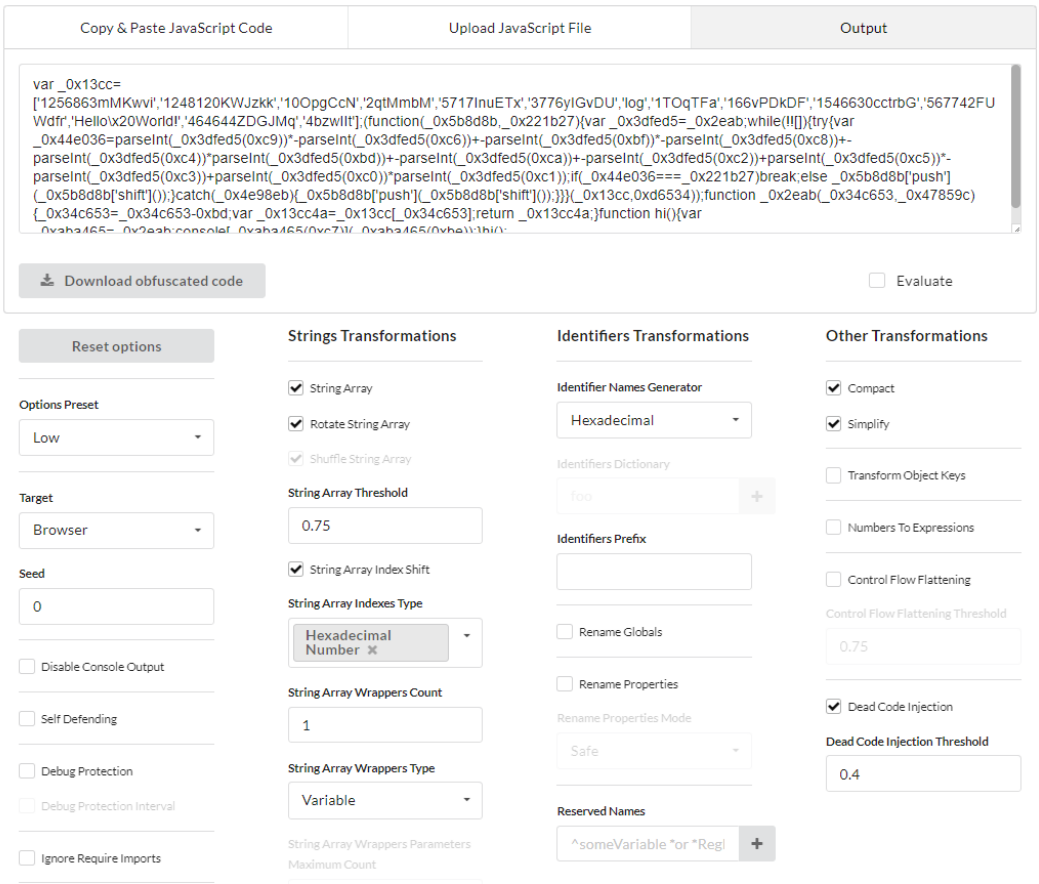
U isječku 3.4 demonstrirane su oba načina obfuskacije logičke strukture.

Isječak 3.4: Primjer obfuskacije logičke strukture

```
//-------------------------------------------------------------
// Početni isječak izvornog koda:
//-------------------------------------------------------------

console.log("Primjer obfuskacije logičke strukture")

//-------------------------------------------------------------
// Isječak koda s umetnutim "mrtvim kodom":
//-------------------------------------------------------------

var i = 1000
if (i < -500) {
        console.log("Umetanje mrtvog koda")
}
console.log("Primjer obfuskacije logičke strukture")

//-------------------------------------------------------------
// Isječak koda s dodanom logičkom strukturom:
//-------------------------------------------------------------

var i = 0;
while (i != 100) {
        if (i == 5) {
                console.log("Primjer obfuskacije logičke strukture")
        }
        i++
}
```

3.5. Obfuskatori izvornog koda

Obfuskaciju se može provoditi ručno, no to je vrlo mukotrpan, spor i neučinkovit način da bi se obfuscirao izvorni kod. U praksi se češće koriste programi koji to rade - obfuskatori. Postoji niz javno dostupnih programa na Internetu koji provode obfuskaciju koda. Oni u pravilu uvijek koriste više navedenih tehnika obfuskacija, te nude razne postavke kojima se može odrediti kakve će se sve transformacije koda provoditi u procesu obfuskacije. Na slici 3.1 je prikazano korisničko sučelje javno dostupnog obfuskatora otvorenog koda obfuscator.io [11]. Tekst u prozoru je obfuscirani JavaScript kod koji ima jednostavnu funkciju ispisivanja "Hello World!". Također su vidljve i razne postavke koje se mogu konfigurirati prije obfuskacije.



Slika 3.1: Javno dostupni obfuskator JavaScript koda obfuscator.io

4. Pregled pristupa detekciji zloćudnog koda

U ovom poglavlju se daje pregled raznih tehnika koje se mogu primijeniti u svrhe detekcije zloćudnog koda. Tehnike će biti opisane i bit će dani neki primjeri sustava koji koriste te tehnike.

4.1. Pristupi koji koriste strojno učenje

Strojno učenje je skup metoda koje u podacima mogu automatski otkrivati obrasce i potom te otkrivene obrasce iskorištavati pri budućem predviđanju podataka [28]. Ideja je na temelju postojećih podataka "naučiti" algoritam strojnog učenja, tj. model da dobro predviđa svojstva novih, još neviđenih podataka. Kod problema detekcije zloćudnog koda, postojeći podaci su primjerci zloćudnog koda, a predviđanje je u stvari određivanje je li neki kod zloćudan ili nije. Poželjno je da model ima dobru sposobnost generalizacije, tj. da dobro nauči gledati bitne stvari u podacima. Kod učenja i predviđanja koriste se značajke. Značajke su svojstva podataka koja će biti promatrana kod učenja modela i predviđanja.

Strojno učenje je postalo vrlo popularna tehnika za rješavanje problema detekcije zloćudnog koda. Ovakvi pristupi često daju bolje rezultate nego drugi pristupi [24].

Prednosti korištenja strojnog učenja je to što detekcija kratko traje, pa su ovakvi pristupi primjereni kada je potrebno analizirati više datoteka. Nedostatci ovakvih pristupa su što neće uvijek detektirati nove primjerke zloćudnog koda ukoliko se oni drastično razlikuju od primjera u skupu za treniranje koje je model vidio, ali i veliki broj lažno pozitivnih rezultata, tj. benignih kodova koji su klasificirani kao zloćudni. Općeniti postupak korištenja strojnog učenja za rješavanje problema detekcije zloćudnog koda se može podijeliti na nekoliko koraka.

Prvi korak je određivanje značajki koje će biti razmatrane pri treniranju modela strojnog učenja. Te značajke mogu biti entropija stringova, prosječna duljina riječi,

dubina i širina sintaksnog stabla i slične vrijednosti.

Nakon što su odabrane i izvučene značajke iz skupa za treniranje, potrebno je odabrati klasifikator i trenirati model. Klasifikator je algoritam koji automatski određuje pripadnost podataka nekoj klasi. U domeni detekcije zloćudnog koda, klase bi bile benigni kod ili zloćudni kod. Neki od klasifikatora koji su se pokazali dobrima za ovu svrhu su slučajne šume, naivan Bayesov klasifikator i metoda potpornih vektora [20] [19] [23].

Nakon učenja, sljedeći korak je vrednovanje modela, tj. određivanje koliko dobro model radi. Ako model ne daje dovoljno dobre rezultate, potrebno je dijagnosticirati zašto, te potom to ispraviti.

Kada su postignuti zadovoljavajući rezultati, model je spreman za instalaciju, odnosno puštanje u produkciju [30].

Primjer postojećeg sustava za detekciju zloćudnog JavaScripta koji koristi strojno učenje je Zozzle [19]. Zozzle kao značajke promatra karakteristike čvorove sintaksnog stabla izvornog koda. Svaka značajka se sastoji od dva svojstva: kontekst u kojem se pojavljuje i tekst sadržan u tom čvoru. Kontekst u ovom slučaju znači u kojoj strukturi programskog jezika se pojavljuje - u petlji, u uvjetu naredbe grananja, *try*/*catch* bloku i slično. Ograničavaju se na promatranja čvorova sintaksnog stabla u kojima se odvija deklaracija varijabli, te na izraze.

Nakon toga promatraju dalje ograniče razmatranje na samo one značajke koje se nakon provođenja statističkih testova pokažu kao najpovezanije s klasifikacijom skripte kao zloćudne ili benigne.

Kao klasifikator strojnog učenja Zozzle koristi Naivan Bayesov klasifikator, koji, pretpostavljajući da su sve značajke nezavisne, računa promatrajući vrijednosti iz skupa za treniranje vjerojatnost da primjer pripada nekoj klasi. Klase su u ovom slučaju zloćudni i benigni program.

Nakon što model obavi učenje, spreman je klasificirati izvorni kod. Kada se izvorni kod preda na analizu u sustav potrebno je ponovno proći sintaksnim stablom koda i odrediti njegove vrijednosti značajki. Izmjerene vrijednosti se zatim koriste u primjeni naivnog Bayesovog klasifikatora kako bi se donijela odluka.

Autori Zozzle-a navode da Zozzle ima iznimno malu stopu lažno pozitivnih rezultata, od samo 0,0003%, te da je vrlo brz i precizan.

4.2. Rješenja koja koriste dinamičku analizu

Dinamička analiza je postupak kojim se promatra ponašanje programa tijekom izvođenja. Program se pokreće u nekoj vrsti izolirane okoline, pa se promatra što se događa dok se program izvodi, npr. prati se koje se funkcije pozivaju, kojim resursima program pristupa, pokušava li program komunicirati u mreži i slično.

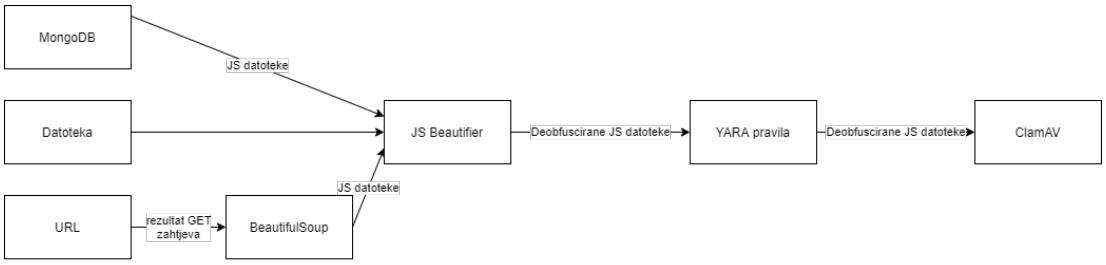
Prednosti dinamičke analize jest to što nije potrebno znati kako izvorni kod izgleda, dakle svejedno je je li kod bio obfusciran ili nije, te obično daje dobre rezultate.

Dinamička analiza, međutim, ima niz problema. Mnogi zloćudni kodovi imaju implementiran neki način otkrivanja jesu li pokrenuti u izoliranom okruženju, pa ako otkriju da jesu neće izvoditi svoju funkcionalnost, izgledat će benigno. Problem je također što dinamička analiza nije skalabilna - potrebno je puno računalnih resursa da se virtualna okruženja u kojima se programi izvode pokreću, te je dinamička analiza spora jer je uvijek potrebno izvršiti kod.

Sustav Expector je primjer rješenja koje koristi dinamičku analizu u svrhu detekcije zloćudnog koda [26]. Konkretan fokus Expectora jest takozvani *malvertising* provođen od dodataka web pregledniku. Funkcionira tako da koristeći Node.js stvore instancu Chromea s dodatkom koji se ispituje, te koristeći Chromeov protokol za udaljeno otklanjanje grešaka u kodu presreću sve pozive JavaScript funkcija virtualnim posrednikom koji koristi Chrome Developer tool. U opisu sustava navode da kako bi procesiranje bilo skalabilno upogonjuju Expector na 60 Linux Debian 7 virtualnih strojeva pokrenutih na poslužitelju s 32 procesorske jezgre i 128 GB RAM-a, dakle relativno velike računalne resurse.

5. Implementirano rješenje

Za implementaciju sustava koji nastoji detektirati zlćudni kod odabran je programski jezik Python 3. Sustav prvo obavlja učitavanje podataka, zatim se nad učitanim podacima obavlja analiza, koja se sastoji od deobfusciranja, primjene YARA pravila, te se koristi ClamAV poslužiteljski proces. Na slici 5.1 prikazane su temeljne komponente sustava, podaci se učitavaju putem URL-a, kao datoteka ili iz MongoDB baze podataka, te se prosljeđuju na analizu ostalim komponentama.



Slika 5.1: Skica komponenti sustava

5.1. Ideja alata

Program je zamišljen kao alat naredbenog retka. Kao inspiracija za odabrani pristup detekcije zlćudnog koda poslužili su dva već postojeća sustava.

YALIH (*Yet Another Low Interaction Honeyclient*) je sustav koji implementira lažnu žrtvu (engl. *honeypot*) koji posjećuje stranice za koje se sumnja da sadrže zlćudni kod kako bi ih se analiziralo [22]. YALIH provodi detekciju zlćudnog koda tako da se kod prvo deobfuscira i zatim predaje na analizu antivirusnim softverima. Korišteni antivirusni softveri zatim grupiraju zlćudne kodove u slične skupine, pa se generiraju YARA pravila specifična za te skupine zlćudnog koda.

WebMon je sustav za detekciju zlćudnog koda koji je temeljen na strojnom učenju i YARA pravilima [21]. WebMon za detekciju obfusciranog koda koristi metode

potpomognute strojnim učenjem, dok neobfuscirani zloćudni kod detektiraju primjenom YARA pravila. Kako bi odredio je li kod obfusciran, WebMon koristi entropiju stringova kao mjerilo.

Implementirano rješenje prvo pokušava otkriti obfuskaciju i po potrebi deobfuscirati izvorni kod koji se analizira. Za samu detekciju su korištena YARA pravila i antivirusni alat otvorenog koda ClamAV [3]. Postupak detekcije je detaljnije opisan u potpoglavlju 5.3.

5.2. Učitavanje podataka

Izvorni kod se programu može predati na analizu u obliku lokalno spremljene datoteke, URL-a web stranice ili zadavanjem URL-a stranice koja je pohranjena u MongoDB bazi podataka Websecradar.

U Websecradar bazi podataka se u kolekciji crawled_data_urls_v0 nalaze URL-ovi raznih web stranica dohvaćenih s Interneta. Preko tih URL-ova se zatim može doći do sadržaja web-stranica kojima URL pripada u kolekciji crawled_data_pages_v0. URL može imati više pripadajućih web-stranica u bazi jer se periodički ponovno dohvaća sadržaj tih stranica s Interneta. Svaki dohvat sadržaja je jedinstveno određen atributom hash, tj. ako dva dohvata imaju isti hash znači da je dohvaćen sadržaj identičan. Program će iz tog razloga provjeravati samo jedan sadržaj po hashu. Za pristup MongoDB bazi podataka iz Pythona korištena je biblioteka pymongo. U isječku 5.1 je prikazana funkcija koja obavlja opisani dohvat, te poziva funkciju koja će provesti detekciju zloćudnog koda.

Isječak 5.1: Učitavanje podataka iz Websecradar baze podataka

```
def load_from_mongodb(url: str):
    client = MongoClient("")    # connection string
    db = client['websecradar']
    url_collection = db['crawled_data_urls_v0']
    document = url_collection.find_one({"url": url})
    checks = document["checks"]
    hash_set = set()
    # scan each check from database, but only once per version
    # (per same hash)
    for check in checks:
        hash_set.add(check['hash'])

    pages_collection = db['crawled_data_pages_v0']
```

```
    for hash in hash_set:
        web_page = pages_collection.find_one({"hash": hash})

        with open('temp', 'w') as f:
            f.write(web_page['page'])
        perform_detection('temp')
```

Različiti načini zadavanja argumenata za ulaz definiraju se zastavicama "-w" ili "–web" za zadavanje URL-a web stranice, "-f" ili "–file" za lokalnu pohranjenju datoteku, te "-m" ili "–mongo" za dohvaćanje iz MongoDB baze podataka. Za dohvaćanje i obradu podataka s web stranica korištene su biblioteke Requests i BeautifulSoup4. Primjer pokretanja programa s web stranicom kao parametrom:

```
$ python malicious_js_detector.py -w https://www.fer.unizg.hr/
```

Biblioteka Requests omogućava jednostavno slanje HTTP zahtjeva na zadani URL, pa je u implementaciji korištena za slanje HTTP GET zahtjeva na URL na kojem se treba provjeriti prisutnost zlćudnog koda.

Kako je cilj rješenja otkriti zlćudni JavaScript kod, a sadržaj dobiven tim GET zahtjevom je HTML dokument, potrebno je izvući JavaScript izvorni kod iz tog HTML dokumenta. Za to je korištena biblioteka BeautifulSoup4 koja omogućava jednostavno parsiranje HTML sadržaja. U implementiranom rješenju je korištena da se iz svih *<script>* oznaka u HTML-u izvuče JavaScript kod, te da se imena JavaScript skripta navedenih u *src* atributu *<script>* spreme u memoriju. Nakon toga se te skripte preuzimaju lokalno koristeći GET zahtjev, kako bi se mogle analizirati. U isječku 5.2 je prikazana funkcija *load_from_webpage()* koja obavlja opisani dohvat podataka i zatim poziva funkciju *perform_detection()* koja provodi detekciju zlćudnog koda.

Isječak 5.2: Funkcija kojom se dohvaćaju podaci s web stranica

```
def load_from_webpage(url: str):
    html_content = requests.get(url).text
    soup = BeautifulSoup(html_content, "html.parser")
    script_files = []
    with open('temp.txt', 'w') as file:
        for tag in soup.select('script'):
            if tag.string is not None:
                file.write(tag.string)
            if tag.attrs.get("src"):
                script_url = urljoin(url, tag.attrs.get("src"))
                script_files.append(script_url)
```

```
js_files = []
for count, script in enumerate(script_files):
    try:
        js_files.append(urllib.request.urlretrieve(script,\
        'temp' + str(count) + '.js')[0])
    except HTTPError:
        print(f"Could not download script {script}")
file = os.path.abspath('temp.txt')
perform_detection(file)
for js_file in js_files:
    perform_detection(js_file)
```

5.3. Provođenje detekcije

Zloćudni kod vrlo je često obfusciran. Iz tog razloga nakon što su podaci učitani i izvorni kodovi izvučeni sljedeći korak je provođenje deobfuskacije izvornog koda. U tu svrhu korišten je alat JS Beautifier [9] koji može deobfuscirati neke javno dostupne obfuskatore koda.

Ukoliko je izvorni kod bio obfusciran, razvijeni sustav ispisuje slabo upozorenje - obfuskacija ne znači nužno da je kod maliciozan. Pokazatelj koji se pokazao preciznim u eksperimentiranju za detekciju obfuskacije bila je entropija stringova, točnije njena razlika prije i nakon provođenja deobfuskacije jer se pokazalo da obfuscirani stringovi obično imaju nižu entropiju [18].

U teoriji informacije, entropija je definirana kao mjera nasumičnosti, tj. mjera informacija sadržanih u nekoj varijabli. Računa se po formuli u nastavku, gdje je $\mathbf{X}$ varijabla koja ima moguće vrijednosti $x_1, x_2, ..., x_n$, a $\mathbf{P}(x_i)$ vjerojatnost pojavljivanja vrijednosti x_i.

$$H(X) = -\sum_{i=1}^{n} P(x_i) log P(x_i)$$

U slučaju analize izvornog koda $\mathbf{X}$ je string, a vrijednosti su svi znakovi koji mogu biti sadržani u stringovima.

S obzirom na razne transformacije moguće transformacije stringova u obfusciranom kodu, ti stringovi će često imati različite entropije od stringova u neobfusciranom kodu. Međutim, znalo se dogoditi u eksperimentiranju da čak i stringovi u neobfusciranim kodovima imaju različite entropije nakon provedene deobfuskacije. Iz tog razloga pri donošenju odluke je li kod bio obfusciran potrebno je imati na umu da razlika entro-

pija ne smije biti zanemarivo mala. Ako se razlike entropije uspoređuju s premalenom konstantom, dolazi do velikog broja lažno pozitivnih rezultata, dok se s prevelikom konstantom češće događa da se obfuskacija ne uspije otrkiti. Kroz isprobavanje se pokazalo da uspoređivanje razlike entropija s 0,65 daje zadovoljavajuće rezultate.

U implementaciji se prije računanja entropije iz koda regularnim izrazima izvlače sadržaji stringova, te se zatim računa njihova entropija. Isječak koda koji obavlja tu funkcionalnost prikazan je u isječku 5.3.

Isječak 5.3: Računanje entropije stingova u Pythonu

```
    strings = re.findall('"[^"]*"', file_bytes)
    strings += re.findall("'.*'", file_bytes)

    if strings:
        entropies = list()
        for string in strings:
            prob = [float(string.count(c)) / \
            len(string) for c in dict.fromkeys(list(string))]
            entropy = - sum([p * log(p) / log(2.0) for p in prob])
            entropies.append(entropy)
        return sum(entropies) / len(entropies)
    else:
        return 0
```

Postupak se dalje nastavlja primjenom YARA pravila na izvorni kod na kojem je provedena deobfuskacija. YARA pravila opisuju postojeći zlćudni kod, dakle potrebno je imati skup primjeraka zlćudnog koda na temelju kojeg se pišu pravila. Za to je korišten GitHub repozitorij JS-Malicious-Dataset [6].

YARA pravila mogu se pisati ručno, ali i generirati koristeći neke javno dostupne alate. Jedan takav alat koji je bio korišten za generiranje YARA pravila za ovaj rad je yarGen [17]. YarGen radi tako da pri generiranju YARA pravila koristi veliki skup stringova koji se pojavljuju u benignim datotekama. Pri analizi zlćudnog koda za kojeg generira pravila tim stringovima se pridaje manja važnost, kako bi pravilo veći utjecaj na klasifikaciju dalo stringovima koji se pojavljuju isključivo u zlćudnom kodu.

Osim generiranih YARA pravila, u implementaciji su korištena i pravila preuzeta s GitHub repozitorija Burp-Yara-Rules [2]. U isječku 5.4 je prikazano jedno od pravila generiranih alatom yarGen.

Isječak 5.4: YARA pravilo generirano alatom yarGen

```
rule prototype_deobfuscated {
    meta:
```

```
        description = "deobfuscated_js_malware - file
                  prototype_deobfuscated.js"
        author = "yarGen Rule Generator"
        reference = "https://github.com/Neo23x0/yarGen"
        date = "2021-05-13"
        hash1 = "504fd62f2be1331ba269fc68b3e7a
                  9645e797b28f8a3c5edbe4ebc9af187a1ec"
    strings:
        $s1 = "  var hosts = new Array('*.postfinance.ch',
        'cs.directnet.com', 'eb.akb.ch', '*.ubs.com', " fullword ascii
        $s2 = "  '*.raiffeisen.ch', '*.credit-suisse.com',
        '*.static-ubs.com', '*.clientis.ch', " fullword ascii
        $s3 = "  for (var i = 0; i < hosts.length; i ++ ){"
        fullword ascii
        $s4 = "function FindProxyForURL(url, host){" fullword ascii
        $s5 = "  var proxy = \"SOCKS 5.34.183.158:80;\";"
        fullword ascii
        $s6 = "  'clientis.ch', '*bcvs.ch', '*.cic.ch',
        'cic.ch', '*baloise.ch', 'ukb.ch', '*.ukb.ch', "
        fullword ascii
        $s7 = "  'tb.raiffeisendirect.ch', '*.bkb.ch',
        'inba.lukb.ch', '*.zkb.ch', '*.onba.ch', " fullword ascii
        $s8 = "  'e-banking.gkb.ch', '*.bekb.ch',
        'wwwsec.ebanking.zugerkb.ch', 'netbanking.bcge.ch', "
        fullword ascii
        $s9 = "  'urkb.ch', '*.urkb.ch', '*.eek.ch', '*szkb.ch',
        '*shkb.ch', '*glkb.ch', '*nkb.ch', " fullword ascii
        $s10 = "      return proxy" fullword ascii
        $s11 = "  return \"DIRECT\"" fullword ascii
        $s12 = "    if (shExpMatch(host, hosts[i])){" fullword ascii
        $s13 = "  '*owkb.ch', '*cash.ch', '*bcf.ch');" fullword ascii
    condition:
        uint16(0) == 0x7566 and filesize < 2KB and
        8 of them
}
```

Za korištenje YARA pravila u Pythonu korištena je biblioteka yara-python. Prvo je potrebno prevesti pravila koja se čitaju iz datoteke koja se zadaje kao argument funkcije *compile()*, u ovom slučaju to je datoteka *yara_js.yar*. Prevođenje pravila se stvara primjerak klase Rules. Traženje podudaranja se izvršava pozivom naredbe *match()* nad stvorenim primjerkom klase Rules. Ukoliko se izvorni kod podudara s YARA pravilom, riječ je o zloćudnom kodu. Opisani postupak je prikazan u isječku 5.5.

Isječak 5.5: Korištenje YARA pravila u Pythonu

```
def yara_scan(file: str):
    rules = yara.compile('yara_js.yar')
    with open(file, 'rb') as f:
        matches = rules.match(data=f.read())
        if matches:
            print("YARA rules have detected malware!")
```

Nakon primjene YARA pravila na izvorni kod, pokreće se ClamAV poslužiteljski proces koji analizira kod i pokušava otkriti je li kod zloćudan.

5.4. Neuspješni pristupi

Osim implementiranog pristupa, isprobani su bili i neki drugi načini detekcije zloćudnog koda koji se nisu pokazali jednako uspješni.

Jedan od isprobanih pristupa je bilo pisanje YARA pravila za obfuscirani zloćudni kod. Ideja je bila uzeti zloćudni kod u izvornom obliku, obfuscirati ga nekim od javno dostupnih JavaScript obfuskatora, pa zatim napraviti YARA pravila koja bi se podudarala s obfusciranim kodom. Međutim, s obzirom na to da postoji jako puno načina da se kod obfuscira, ovaj način se pokazao neučinkovitim. Naime, javno dostupnih obfuskatora je mnogo i nude razne opcije. Obfuscirani kod koji se dobije se uvijek drastično razlikuje. Kako bi se tim pristupom pokrio velik broj mogućih obfusciranih kodova, bilo bi potrebno za svaki primjerak zloćudnog koda generirati veoma puno pravila. Također, napadači mogu koristiti obfuskatore koji nisu javno dostupni i dobiti do sada neviđen oblik zloćudnog koda koji nije bilo moguće opisati YARA pravilima. Ukratko, ovaj pristup bi bio veoma spor i vrlo neučinkovit.

Slična ideja je bila da se svi dostupni primjerci zloćudnog koda obfusciraju istim obfuskatorom, te da se zatim pišu YARA pravila koja bi opisivala kod u obfusciranom obliku. Za ovaj pristup je trebalo isprobati hoće li napravljena pravila prepoznavati kod koji je potencijalno obfusciran nekim drugim obfuskatorom kao maliciozna, nakon što se taj isti kod obfuscira obfuskatorom koji je korišten pri stvaranju pravila. Ovaj pristup se također pokazao neuspješnim zato što se izvorni kodovi koji su bili prethodno obfuscirani nakon obfuskacije obfuskatorom korištenim pri pisanju pravila nisu podudarali s dobivenim pravilima. Uspješno su bili detektirani neobfuscirani primjerci zloćudnog koda, no to se postiže primjenom YARA pravila i bez obfuskacije, pa je obfuskacija suvišna u ovom postupku.

5.5. Evaluacija implementiranog rješenja

Implementirano rješenje dobro funkcionira na primjercima neobfusciranog zloćudnog koda koji su dostupni na Internetu, takve primjerke YARA pravila uspješno detektiraju. Također funkcionira i kada se ti zloćudni kodovi obfusciraju nekim lošijim javno dostupnim obfuskatorima čije obfuskacije alat JS Beautifier može deobfuscirati. YARA pravila ne mogu detektirati dobro obfuscirani kod, niti primjerke zloćudnog koda za koja ne postoje napisana pravila.

Rješenje također pronalazi sve zloćudne kodove koje ClamAV ima u svojoj bazi podataka, tj. može detektirati. U slučajima kada analizirani zloćudni kod nije opisan korištenim YARA pravilima, rješenje se oslanja na ClamAV za detekciju. ClamAV je testiran na primjercima zloćudnog koda na GitHub repozitoriju github.com/HynekPetrak/javascript-malware-collection, uspješno je prepoznao 77,14% zloćudnih datoteka.

Na nekim primjercima iz tog istog GitHub repozitorija je testirana i detekcija obfuskacije. Testirano je i na raznim benignim JavaSript kodovima (npr. iz biblioteke JQuery [5]) obfusciranim raznim javno dostupnim obfuskatorima. Obfuskacija je detektirana u većini slučajeva koja je bila prisutna, problematična je relativno visoka stopa lažno pozitivnih rezultata od oko 10

Detekcija je pokrenuta na otprilike 20 legitimnih web stranica na Internetu. Tu su bile uključene razne popularne društvene mreže, stranice Sveučilišta u Zagrebu i pripadnih fakulteta, stranica s vremenskom prognozom i slične. Detekcija je također pokrenuta i na otprilike 20 nasumično odabranih stranica iz Websecradar baze podataka. Nije bio detektiran zloćudni kod i nije bilo lažno pozitivnih rezultata.

Problemi ovog pristupa su nemogućnost klasificiranja novih primjeraka zloćudnog koda koji nisu opisani postojećim YARA pravilima i koji se ne nalaze u bazi podataka ClamAV-a, te učinkovita obfuskacija koda. S obzirom na to da rješenje nije učinkovito kada je u slučaju kvalitetno obfusciran zloćudni kod, obfuskacija se nastoji detektirati tako da korisnik ima tu informaciju, pa onda može drukčije postupati sa stranicom ili ju potpuno izbjeći ukoliko nije siguran u njezinu vjerodostojnost.

6. Zaključak

Razvijeno rješenje predstavlja jedan pristup detekciji zloćudnog koda, te pokazuje koliko su YARA pravila koristan alat za tu svrhu. YARA pravila su davala dobre rezultate za neobfuscirani zloćudni kod.

Obfuskacija zloćudnog koda je problem koji iznimno otežava statičku analizu zloćudnog JavaScript koda. Isprobani su pristupi pisanja YARA pravila za obfuscirani zloćudni kod. Jedan pristup je bio pisanje YARA pravila za javno dostupan obfuscirani zloćudni kod. Drugi je pristup bio da se pri pisanju pravila kod dodatno obfuscira nekim obfuskatorom, pa da se pri provjeri kod koji se analizira obfuscira tim istim obfuskatorom i zatim uspoređuje s napisanim YARA pravilima. Oba pristupa su se pokazala neuspješnima, što upućuje na to da bi za obfuscirani kod vjerojatno bilo bolje koristiti neku drugu metodu detekcije.

Međutim, neke metode koje koriste strojno učenje su se pokazale obećavajućima kako bi se riješio problem detekcije obfusciranog zloćudnog koda statičkom analizom. Dinamička analiza je također dobra metoda za detekciju zloćudnog koda, no spora je, pa neki akademici predlažu i zagovaraju razne hibridne pristupe koji koriste i statičku i dinamičku analizu koda u svrhu detekcije zloćudnosti.

LITERATURA

[1] Angler exploit kit. https://www.theregister.com/2016/08/16/angler_8734564567/. 30.6.2021.

[2] Github repozitorij burp-yara-rules. https://github.com/codewatchorg/Burp-Yara-Rules. 3.6.2021.

[3] Clamav. https://www.clamav.net/. 3.6.2021.

[4] Drive-by compromise. https://attack.mitre.org/techniques/T1189/. 9.6.2021.

[5] Jquery. https://github.com/jquery/jquery], note = 30.6.2021.

[6] Malicious javascript dataset. https://github.com/geeksonsecurity/js-malicious-dataset, . 28.5.2021.

[7] Javascript malware collection. https://github.com/HynekPetrak/javascript-malware-collection, . 28.5.2021.

[8] Usage statistics of javascript. https://w3techs.com/technologies/details/cp-javascript/, . 28.5.2021.

[9] Js beautifier. https://pypi.org/project/jsbeautifier/. 9.6.2021.

[10] Malvertising. https://www.cisecurity.org/blog/malvertising/. 9.6.2021.

[11] Javascript obfuscator tool - obfuscator.io. https://obfuscator.io/. 28.5.2021.

[12] Internet usage worldwide - statistics & facts. https://www.statista.com/topics/1145/internet-usage-worldwide/. 28.5.2021.

[13] Exploit kits. https://www.trendmicro.com/vinfo/us/security/definition/exploit-kit. 30.6.2021.

[14] Wordpress.

[15] Cross site scripting (xss). https://owasp.org/www-community/attacks/xss/. 9.6.2021.

[16] Yara rules documentation. https://yara.readthedocs.io/en/stable/index.html, . 3.6.2021.

[17] yargen - yara rule generator. https://github.com/Neo23x0/yarGen, . 3.6.2021.

[18] Davide Canali, Marco Cova, Giovanni Vigna, i Christopher Kruegel. Prophiler: a fast filter for the large-scale detection of malicious web pages. U *Proceedings of the 20th international conference on World wide web*, stranice 197–206, 2011.

[19] Charlie Curtsinger, Benjamin Livshits, Benjamin G Zorn, i Christian Seifert. Zozzle: Fast and precise in-browser javascript malware detection. U *USENIX security symposium*, stranice 33–48. San Francisco, 2011.

[20] Aurore Fass, Robert P Krawczyk, Michael Backes, i Ben Stock. Jast: Fully syntactic detection of malicious (obfuscated) javascript. U *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, stranice 303–325. Springer, 2018.

[21] Sungjin Kim, Jinkook Kim, Seokwoo Nam, i Dohoon Kim. Webmon: Ml-and yara-based malicious webpage detection. *Computer Networks*, 137:119–131, 2018.

[22] Masood Mansoori, Ian Welch, i Qiang Fu. Yalih, yet another low interaction honeyclient. U *Proceedings of the Twelfth Australasian Information Security Conference-Volume 149*, stranice 7–15, 2014.

[23] Kristof Schütt, Marius Kloft, Alexander Bikadorov, i Konrad Rieck. Early detection of malicious behavior in javascript code. U *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, stranice 15–24, 2012.

[24] Md Fahimuzzman Sohan i Anas Basalamah. A systematic literature review and quality analysis of javascript malware detection. *IEEE Access*, 8:190539–190552, 2020.

[25] Junjie Wang, Yinxing Xue, Yang Liu, i Tian Huat Tan. Jsdc: A hybrid approach for javascript malware detection and classification. U *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, stranice 109–120, 2015.

[26] Xinyu Xing, Wei Meng, Byoungyoung Lee, Udi Weinsberg, Anmol Sheth, Roberto Perdisci, i Wenke Lee. Understanding malvertising through ad-injecting browser extensions. U *Proceedings of the 24th international conference on world wide web*, stranice 1286–1295, 2015.

[27] Wei Xu, Fangfang Zhang, i Sencun Zhu. The power of obfuscation techniques in malicious javascript code: A measurement study. U *2012 7th International Conference on Malicious and Unwanted Software*, stranice 9–16. IEEE, 2012.

[28] Marko Čupić. *Umjetna inteligencija : Uvod u strojno učenje (skripta)*. 2020.

[29] Ante Đerek, Stjepan Groš, Miljenko Mikuc, i Marin Vuković. Zloćudni kod. prezentacija s kolegija Sigurnost računalnih sustava na FER-u, 2021.

[30] Jan Šnajder. Strojno učenje: Osnovni koncepti. skripta s kolegija Strojno učenje na FER-u, 2020.

Tehnike otkrivanja zloćudnog kôda na web stranicama

Sažetak

Internet je u današnje vrijeme vrlo raširen i vrlo je povoljan za širenje zloćudnog koda. U ovom radu je dan pregled raznih vrsta zloćudnog koda koje postoje, te su objašnjene relevantne specifičnosti programskog jezika JavaScript koji je kod gotovo svih web stranica korišten za razvoj klijentske strane. Razvijen je sustav koji nastoji otkriti zloćudan JavaScript kod korištenjem antivirusnog alata otvorenog koda ClamAV i YARA pravila, koja su alat za opis konkretnih primjeraka ili obitelji zloćudnog koda. Opisana je obfuskacija, postupak koji najviše otežava detekciju zloćudnog koda, te su isprobani neki eksperimentalni pristupi korištenja YARA pravila s ciljem detekcije obfusciranog zloćudnog koda. Dan je pregled raznih pristupa problemu detekcije zloćudnog JavaScript koda i navedene su njihove prednosti i nedostaci.

Ključne riječi: Statička analiza, YARA pravila, kibernetička sigurnost, malware.

Techniques for Detecting Malware on Websites

Abstract

The Internet is very widespread today and is favorable as a platform for spreading malware. In this paper an overview of various malware kinds is given, and some relevant specific characteristics of the JavaScript programming language are explained. JavaScript is important for this topic because it is used on a large majority of web pages on the client side. A system that uses ClamAV, an open-source antivirus tool, and YARA rules, which are a tool for describing specific malware samples or families, to detect malicious JavaScript code is implemented. A description of obfuscation is given, since it represents the biggest problem in malware detection, as well as a description of some experimental approaches to detecting obfuscated malicious JavaScript code using YARA rules that were tried out. Various methods of detecting malicious JavaScript codes are explained, as well as their advantages and drawbacks, accompanied by concrete examples.

Keywords: Static analysis, YARA rules, cybersecurity, malware.