CARNET

Traženje ranjivosti softvera alatima za statičku analizu kôda

CERT.hr-PUBDOC-2019-12-393

CERT.hr

Sadržaj

Ovaj dokument izradio je Laboratorij za sustave i signale Zavoda za elektroničke sustave i obradbu informacija Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

1 Uvod

Razvojem modernih tehnologija i prodorom računala u sve aspekte ljudskoga života, čovjek je postao okružen računalnim programima i ovisan o njima u gotovo svim svojim djelatnostima. Naše znanje, financije, medicinski podaci pa i naša društvena komunikacija povjereni su računalima i ovisni o njihovom ispravnom radu. Zbog ovako velikog utjecaja računala i interneta na našu svakodnevicu, lako je razumjeti želju i potrebu da se privatni podaci adekvatno zaštite. Neovlaštenim pristupom financijskim ili drugim povjerljivim podacima, zlonamjerni napadač mogao bi počiniti značajnu štetu. Kad govorimo o tvrtkama i poslovnim subjektima, štete se mogu mjeriti u milijunima, a nerijetko je ugrožena i reputacija tvrtke i povjerenje klijenata.

U računalnoj terminologiji, ranjivost je pogreška u programiranju, postavkama ili primjeni komponenti računalnog sustava, tj. slabost koju napadač može iskoristiti za obavljanje nedozvoljenih radnji na računalnom sustavu. Nedozvoljene radnje mogu uključivati neovlašteni pristup, neovlašteno dohvaćanje podataka ili čak izvršavanje proizvoljnog računalnog kôda na sustavu. Ovakva rječnička definicija, međutim, ne daje dovoljno slikovit uvid u pravu opasnost ranjivosti i njihovu široku rasprostranjenost, ostavljajući mogućnost tumačenja da se od ranjivosti trebaju štititi samo državne službe i bankarske institucije dok prosječne tvrtke nemaju razloga za zabrinutost jer koriste samo jednostavne računalne programe.

Stvarnost je, nažalost, potpuno suprotna. Razvoj računalnih programa i *online* aplikacija veći je i brži nego ikad zbog lake dostupnosti i financijske pristupačnosti tehnologije i infrastrukture. To dovodi do sve veće informatizacije društva, čime se povećava i kvaliteta ljudskog života, jer se sve više obaveza i aktivnosti može riješiti pomoću svog računala ili mobitela. S druge strane, razvoj softvera se sve više povjerava manje iskusnim izvođačima koji u međusobnom tržišnom natjecanju često žrtvuju kvalitetu i temeljitost u korist niže krajnje cijene i zadovoljavanja sve kraćih rokova isporuke.

Da računalna ranjivost može ugroziti poslovanje i tvrtke koja naizgled uopće ne ovisi o informacijskim tehnologijama vidi se na primjeru tvrtke *Panera Bakery*, lanca restorana prvenstveno pekarskih proizvoda. Dylan Houlihan, istraživač iz područja računalne sigurnosti, u kolovozu 2017. otkrio je ranjivost u njihovom aplikacijskom programskom sučelju zaduženom za obradu narudžbi (1). Ispostavilo se da je jednostavnim upisom određenog URL-a u web preglednik moguće saznati puno ime i prezime, adresu, email adresu, korisničko ime, broj telefona, datum rođenja, prehrambene navike kao i posljednje četiri znamenke kreditne kartice bilo kojeg korisnika. Još gore, identifikacijski broj korisnika unutar aplikacije je slijedni broj, pa je njegovim jednostavnim inkrementalnim uvećavanjem moguće dohvatiti podatke za sve kupce te tvrtke. Nažalost, *Panera Bakery* je mjesecima ignorirala dojavu o ranjivosti što je u konačnici izazvalo revolt javnosti i veliku štetu ugledu tvrtke.

Razvoj softvera nije savršen proces. Programeri rade pritisnuti kratkim rokovima i uz nedostatak ljudskih resursa, često i s najnovijim, nedovoljno poznatim i provjerenim alatima, pa je potpuno normalno očekivati pogreške i propuste. Ovo je jednostavna činjenica i svaka ozbiljna softverska tvrtka s time mora računati – izvorni se kôd jednostavno mora provjeriti prije nego se proglasi spremnim za široku uporabu.

Provjereni i pouzdani način otkrivanja pogrešaka je uzajamna provjera kôda (engl. *peer review*) pri čemu programeri analiziraju izvorni kôd koji su napisali njihovi kolege. Pažljivim čitanjem kôda kolege programeri s vrlo visokim postotkom pouzdanosti otkrivaju pogreške koje je sam autor kôda previdio. Pri tome je dobar običaj da autor kôda ne objašnjava niti pojašnjava svoj kôd. Sam kôd, naime, mora biti dovoljno jasan i razumljiv kako u budućnosti ne bi postao mogući izvor pogrešaka. Kôd koji ne zadovoljava ovaj osnovni kriterij potrebno je prepraviti i učiniti čitkijim.

Ovakva provjera kôda odlično funkcionira, ali je i izuzetno skupa. Osim pisanja izvornog kôda, programeri tada moraju redovito provjeravati softverski kôd svojih kolega. To se mora ponavljati za svaki novi dio kôda, ali također i nakon svake prepravke postojećih dijelova kôda. Uz to, programerima je nužno osigurati dovoljno odmora kako bi se izbjegao zamor i gubitak pažnje, čime bi sama provjera postala besmislena. U današnje vrijeme ubrzanih rokova i smanjenih raspoloživih resursa, uzajamna provjera kôda često postaje nepraktična i nemoguća.

Stoga je normalna i očekivana težnja da se provjera kôda povjeri računalu. Jedno kompromisno rješenje je uporaba alata za statičku analizu kôda. Ovim se alatima automatizira jedan dio provjere kôda i priprema posao za kasniju ljudsku analizu. Za razliku od dinamičke analize programa u izvršavanju, statička analiza provjerava tekstualni programski kôd bez njegovog pokretanja na računalu.

Zbog lakoće primjene, statička se analiza kôda može uključiti u proces razvoja softvera od njegovog samog početka, čime se ostvaruju značajne uštede u ukupnom potrebnom trudu jer je svaku grešku teže ispraviti što je ona kasnije otkrivena. Alatima za automatsku provjeru kôda moguće je svakodnevno provjeravati cjelokupni kôd i otkriti greške čim nastanu. A budući da, osim pogrešaka, ovi alati mogu ukazati i na nedosljednosti ili nedostatke u formatiranju kôda, moguće je postići poštivanje i pridržavanje najboljih praksi već od samog početka razvoja. Ovo ih čini korisnima i kao edukativno sredstvo za nove programere koji još nisu u potpunosti usvojili pravila programiranja tvrtke.

2 Što je statička analiza kôda

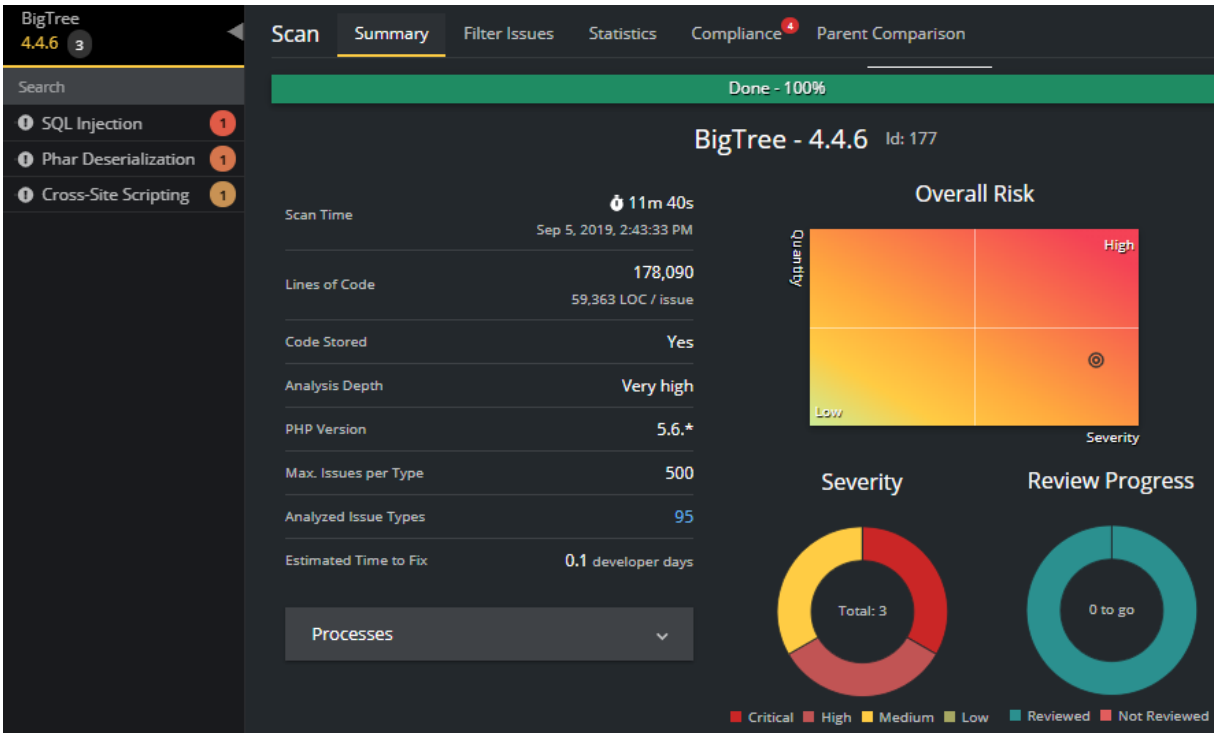
Statička analiza programa (tj. programskog kôda) je analiza koja se provodi bez da se stvarno pokrene (izvrši) program. Osim statičke, postoji još i dinamička analiza koja se provodi nad programom u izvršavanju.

Pojmovi statičke i dinamičke analize mogu se odnositi na traženje bilo kakvih grešaka u programskom kôdu, ili još općenitije, na bilo kakvu analizu kôda koja se provodi bez pokretanja odnosno s pokretanjem programa. Primjerice, ti pojmovi se koriste i u kontekstu provjere ispravnosti programa, i u kontekstu detekcije zlonamjernog kôda (npr. kao dio antivirusnog programa). Također, pojmovi statičke i dinamičke analize se ne odnose samo na automatizirano testiranje. Primjerice, ručna provjera u kojoj osoba čita programski kôd kako bi našla greške će se isto nazivati statičkom analizom. Slično tome, ručna provjera u kojoj osoba pokreće program, i kroz korištenje programa pokušava otkriti greške, će se nazivati dinamičkom analizom.

Ipak, fokus ovog dokumenta bit će na traženju ranjivosti u programskom kôdu alatima za statičku analizu kôda. Takva analiza detaljno ispituje programski kôd kako bi detektirala moguće sigurnosne ranjivosti, a izvodi se automatizirano različitim softverskim alatima. Korištenjem takvih specijaliziranih alata može se djelomično automatizirati sigurnosno testiranje koje se može pokretati više puta i nakon svake izmjene u kôdu. Takva statička analiza pronaći će pogreške u kôdu, tj. ranjivosti koje bi napadač mogao iskoristiti za napad. Neke od uobičajenih ranjivosti koje se javljaju u programskom kôdu su umetanje kôda/naredbi, prelijevanje međuspremnika, itd.

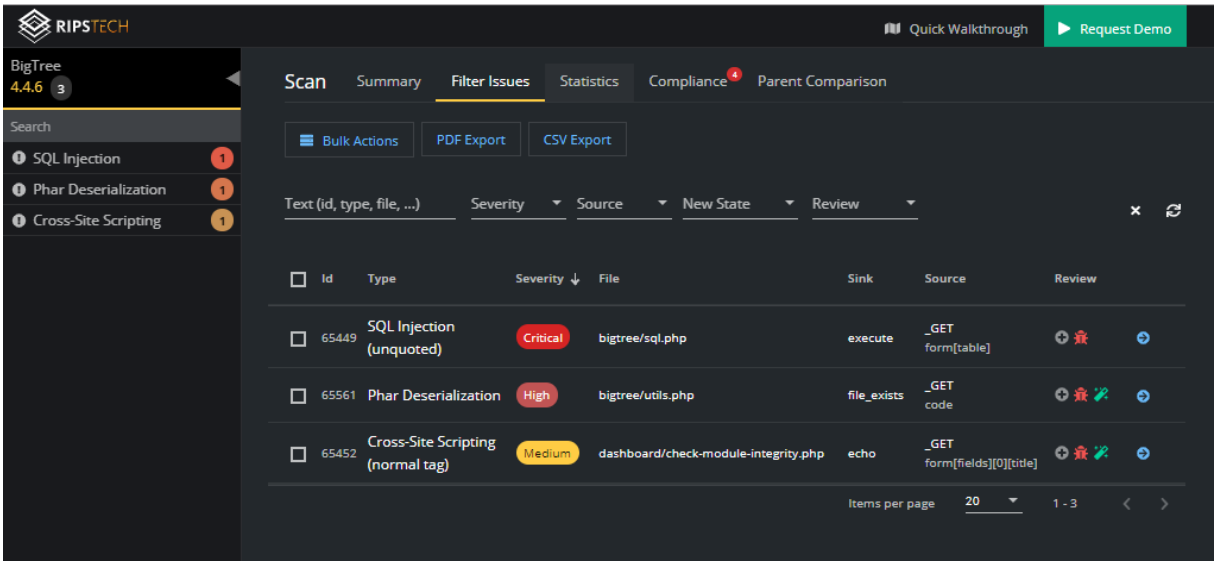
Organizacija OWASP je sastavila popis alata za ispitivanje ranjivosti statičkom analizom kôda za određene programske jezike na ovoj poveznici. Jedan od takvih alata je i RIPS, namijenjen analizi softvera napisanog u programskim jezicima PHP ili Java. U ovom dokumentu će na primjeru RIPS-a biti opisane uobičajene mogućnosti alata za statičku analizu kôda.

Alati poput RIPS-a omogućavaju korisnicima jednostavan (i grafički i deskriptivni) pregled rezultata nakon statičke analize. Na slici 1 prikazano je sučelje RIPS-a *Summary* na kojem je prikazan sažetak rezultata posljednje statičke analize softvera. Osim osnovnih informacija o vremenu analize, broju analiziranih linija kôda i pronađenih ranjivosti, RIPS također procjenjuje koliko će vremena biti potrebno da programer popravi problem.



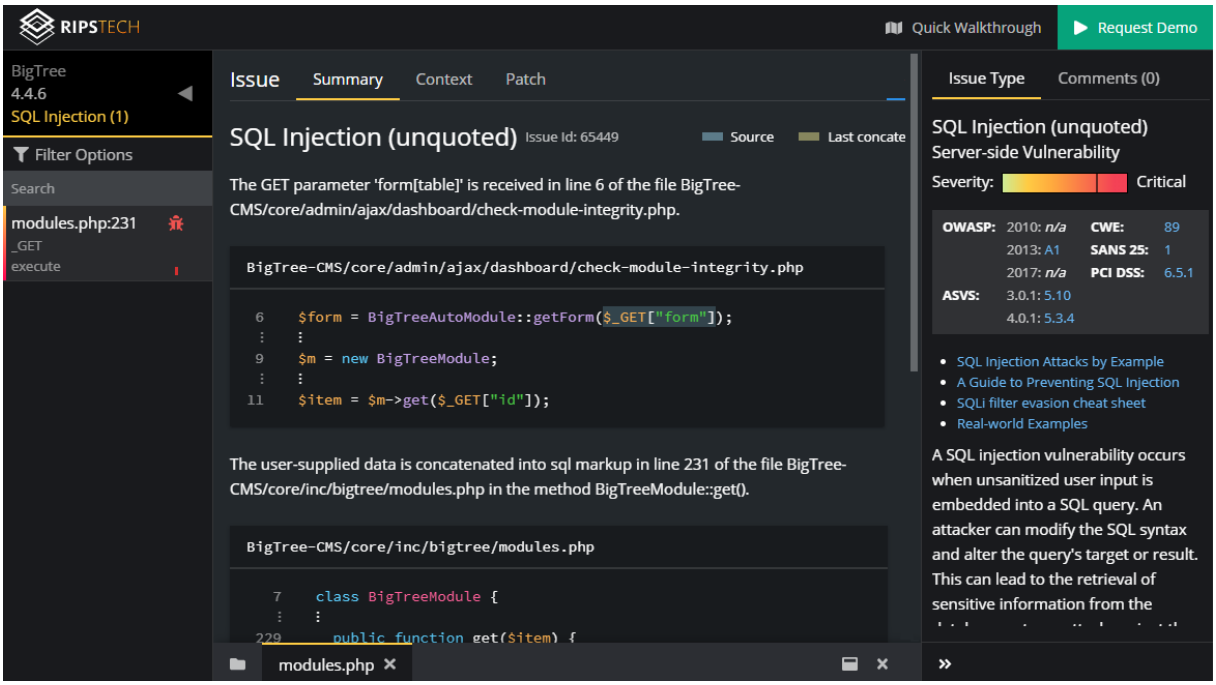
Slika 1 ***Sučelje Summary sa sažetkom rezultata posljednje statičke analize alatom RIPS*** **(2)**

Sučelje *Filter Issues* prikazat će korisniku sve ranjivosti na koje je alat naišao i poredati ih po rizičnosti kao što je prikazano na slici 2.



Slika 2 ***Sučelje Filter Issues s popisom svih pronađenih ranjivosti poredanih po rizičnosti*** **(2)**

Klikom mišem na bilo koju od pronađenih ranjivosti otvara se sučelje s objašnjenjem gdje je pogreška pronađena u programskom kôdu i isticanjem tih linija kôda. Također, na desnoj strani prozora nalazi se i kratko objašnjenje o pronađenoj ranjivosti, primjeri napada koji iskorištavaju tu ranjivost, upute za popravljanje pogreške i općenito objašnjenje zašto je takva pogreška rizična za aplikaciju i kako je napadač može iskoristiti kao ranjivost.



Slika 3 ***Detalji pojedine ranjivosti*** **(2)**

Sučelje *Statistics* prikazuje razne statistike o pronađenim ranjivostima, npr. koliko njih spada u kritične, koliko u visoke, a koliko u srednje i nisko rizične ranjivosti.

Slika 4 ***Sučelje Statistics s raznim statistikama o pronađenim ranjivostima*** **(2)**

Ovakav alat koristan je jer programeri jednim pogledom mogu utvrditi koje su potencijalne ranjivosti pronađene i brzo početi s popravljanjem programskog kôda. Alat će im poredati pronađene pogreške po prioritetu rješavanja i uputiti ih na to kako te pogreške popraviti. Programer može slijediti upute i poveznice koje mu alat nudi kako bi popravio kôd, ili može prepustiti RIPS-u da sam generira sigurnosnu zakrpu na način da automatski ispravi kritične dijelove kôda.

Na slici 5 prikazane su linije kôda u kojima je RIPS statičkom analizom detektirao ranjivost.

Controller/UserController.php

```
10 class UserController extends AbstractController
11 {
12   public function show(Request $request): Response
13   {
14     $name = $request->get('name');
 :     :
19     $image = "<img title='" . $name . "' src='1.png'/>";
 :     :
26     return new Response($image);
27   }
28 }
```

Slika 5 ***Linije programskog kôda u kojima je RIPS detektirao ranjivost*** **(3)**

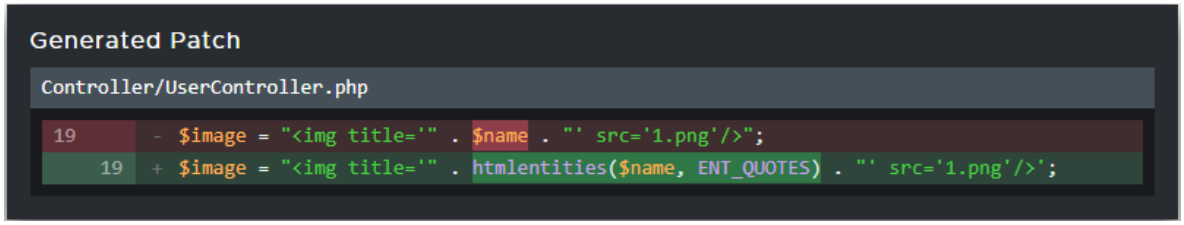
U tim linijama kôda ranjivost je to što se ni u jednom trenutku ne provjerava tekst (naslov slike) kojeg je unio korisnik. Korisnikovom unosu nikad ne treba vjerovati jer napadač može pokušati podmetnuti posebno konstruirani tekst kojeg će web preglednik izvršiti kao kôd, što može rezultirati napadom na žrtvin korisnički računa na web stranici ili na njeno računalo. Kao što je prikazano na slici 6, RIPS istakne dio kôda koji treba prepraviti i izvješćuje korisnika o kakvom napadu može doći zbog tako napisanog kôda (u ovom konkretnom slučaju to bi bio XSS napad o kojemu je više moguće pročitati u dokumentima nacionalnog CERT-a „Sigurnosni rizici JavaScript kôda prilikom pregledavanja weba“).

Cross-Site Scripting Vulnerability
(single-quoted attribute)

```
<img title='  $_REQUEST['name']  ' src='1.png'/>
```

Slika 6 ***RIPS izvješćuje programera o dijelu kôda kojeg je potrebno ispraviti*** **(3)**

No, alat može i sam generirati sigurnosnu zakrpu (engl. *Generate Patch*) na način da automatski ispravi pogrešku tako da doda PHP funkciju *htmlentities()* kao što je prikazano na slici 7. Ta će funkcija spriječiti XSS napad jer neće dopustiti da se korisnikov unos interpretira kao kôd, već će problematične znakove pretvoriti u bezopasan znakovni niz.



Slika 7 ***RIPS sam generira sigurnosnu zakrpu, tj. dodaje potrebnu funkciju u kôd*** **(3)**

RIPS je samo jedan primjer korisnog alata, no bitno je razumjeti načela, tj. metode na kojima počivaju RIPS i njemu slični alati za statičku analizu kôda kako bismo razumjeli kako uopće takvi alati uspiju detektirati ranjivosti i staviti ih u kontekst. U nastavku ovog dokumenta objasnit će se metode, prednosti i nedostaci ovakvog pristupa traženju ranjivosti.

2.1. Metode statičke analize kôda

U provjeri izvornog kôda, alati za statičku analizu kôda primjenjuju različite metode i tehnike. Neke od najčešćih i najuobičajenijih su:

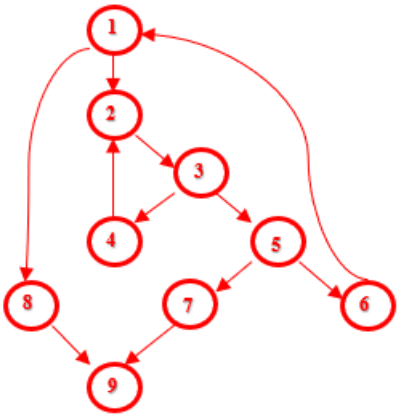
- Analiza toka podataka koja prikuplja informacije o skupovima mogućih vrijednosti u različitim točkama izvršavanja programa,
- Analiza zagađenja koja provjerava koje sve varijable mogu biti „zagađene“ korisničkim unosom i prati njihovu uporabu kroz funkcije programa
- Leksička analiza koja raščlanjuje tekst izvornog kôda u informacijske tokene kako bi računalo moglo razumjeti njegovo značenje.

2.1.1 Analiza toka podataka

Analiza toka podataka (engl. *Data Flow Analysis*) je tehnika prikupljanja informacija o skupovima mogućih vrijednosti u različitim točkama izvršavanja programa. Izvorni se kôd najprije podijeli u osnovne blokove – skupove uzastopnih naredbi koje imaju samo jedan ulaz na početku bloka, samo jedan izlaz na kraju bloka te unutar bloka ne dopuštaju grananje ili zaustavljanje izvršavanja programa. Tijek izvršavanja programa se zatim predstavi grafom toka kontrole čiji čvorovi predstavljaju osnovne blokove povezane usmjerenim točkama skoka u kojima tijek izvršavanja prelazi s jednog bloka na drugi.

Postoji nekoliko metoda obavljanja analize toka podataka, no najjednostavnija je postavljanje jednadžbi toka podataka za svaki čvor grafa toka kontrole i njihovo rješavanje uzastopnim računanjem izlaza u ovisnosti o ulazima za svaki čvor zasebno. Ovaj se proces ponavlja dok se ne dosegne stabilno stanje cijelog sustava.

```
int binaryfind(int x, int v[], int n)
{
  int low, high, mid;
  low = 0;                          1
  high =  n - 1;
  while (low <= high)               2
  {
    mid = (low + high)/2;           3
    if (x < v[mid])
      high = mid - 1;               4
    else if (x > v[mid])            5
      low = mid + 1;                6
    else return mid;                7
  }
  Return -1;                        8
}                                   9
```



Slika 8 ***Primjer grafa kontrole toka***

2.1.2 Analiza zagađenja

Analiza zagađenja (engl. *Taint Analysis*) provjerava koje sve varijable mogu biti „zagađene“ korisničkim unosom i prati njihovu uporabu kroz funkcije programa. Svaka varijabla korisničkog unosa koja prije uporabe nije pročišćena, tj. valjanost njenog sadržaja nije provjerena, može potencijalno uzrokovati ranjivost. Osnovno je pravilo da se korisničkom unosu ne smije vjerovati i većina ranjivosti i sigurnosnih prijetnji danas potječe upravo od nepravilnog baratanja podacima koje unose korisnici programa čime se otvaraju vrata umetanju SQL upita, XSS napadu, umetanju kôda, prelijevanju međuspremnika i mnogim drugim.

Na slici 9 prikazana je analiza zagađenja dodatkom (engl. *Extension*) za *Visual Studio* koji statički analizira programski kôd napisan u programskom jeziku C# i razvojnoj okolini .NET. Rezultat analize zagađenja su tri ispravno sastavljena dijela upita koji će biti proslijeđen bazi podataka i jedan neispravan dio koji bez provjere spaja (engl. *concatenates*) korisnikov unos poštanskog broja s ostatkom upita, zbog čega može doći do umetanja SQL upita (engl. *SQL injection*). Drugim riječima, varijabla *Query* u tom slučaju može biti zagađena korisničkim zlonamjernim unosom.

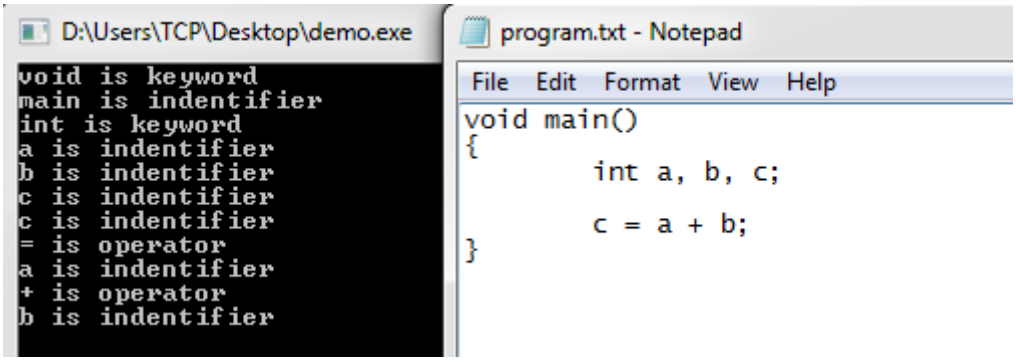
```
0 references | 0 changes | 0 authors, 0 changes
public static IEnumerable<UserEntity> GetUsers(DataContext ctx, string PostalCode,
    bool ActiveUser, bool InMontreal)
{
    var Query = @"SELECT UserID, FirstName, LastName, Address, City, Country
        FROM Users";
    Query += "WHERE  PostalCode = '"+PostalCode+"'";
    if (ActiveUser) {
        Query += " AND Active = true";
    }
    if (InMontreal) {
        Query += " AND City = 'Montreal'";
    }
    return ctx.ExecuteQuery<UserEntity>(Query);
}
```

Slika 9 ***Rezultati analize zagađenja alatom Roselyn nad C# programskim kôdom*** **(4)**

2.1.3 Leksička analiza

Leksička analiza (engl. *Lexical Analysis*) je metoda koja ne služi izravno za pronalazak ranjivosti, već se koristi kao priprema za druge metoda. Leksička analiza raščlanjuje tekst izvornog kôda u informacijske tokene kako bi računalo moglo razumjeti njegovo značenje. Token je niz znakova s poznatim značenjem kao npr. ključna riječ, broj ili operator. Leksička analiza jedan je od prvih koraka prevođenja (engl. *compile*) izvornog kôda i, premda u kontekstu statičke analize kôda ona ne pronalazi ranjivosti sama po sebi, ona je korisna kao priprema ostalim metodama analize.

Na slici 10 prikazan je primjer programa za pojednostavljenu leksičku analizu koja raspoznaje ključne riječi, operatore i identifikatore u programskom jeziku C/C++.



Slika 10 ***Pojednostavljena leksička analiza C/C++ programa*** **(5)**

2.1. Prednosti statičke analize kôda

Temeljna prednost statičke analize kôda je omogućavanje velikih ušteda tijekom razvoja programske podrške ranim i redovitim otkrivanjem pogrešaka. Prema podacima koje navodi Steve McConnell u svojoj knjizi „*Code complete*“, ispravljanje pogreške otkrivene u završnoj fazi testiranja softvera deseterostruko je skuplje od ispravljanja pogreške otkrivene u fazi razvoja (6). A trošak ispravljanja pogreške otkrivene nakon izdavanja softvera može biti višestruko viši i od toga. Alati za statičku analizu kôda mogu otkriti brojne pogreške i nedosljednosti još u ranim fazama razvoja i time ostvariti višestruke uštede.

Osim financijske i vremenske uštede, još neke od prednosti su:

- **Potpuna pokrivenost kôda.** Statičkom analizom provjeravaju se svi dijelovi kôda, uključujući i one dijelove koji se rijetko provjeravaju drugim metodama ili je njihovo testiranje zahtjevno i komplicirano. Na ovaj je način moguće otkriti npr. pogreške u rutinama za upravljanje iznimkama (engl. *exception handlers*) ili u sustavu za bilježenje dnevnika (engl. *log*).

- **Neovisnost o prevodiocu i razvojnoj okolini.** Statička analiza provjerava izvorni kôd na sličan način kako to čine sami programeri – analizom samog teksta. Ovaj je način provjere u potpunosti neovisan od razvojne okoline u kojoj je izvorni kôd napisan kao i od prevodioca i ostalih alata koji se koriste u izgradnji završnog softverskog paketa. Neovisnom provjerom moguće je otkriti greške uzrokovane nedefiniranim ponašanjem, a koje bi se u suprotnom pojavile u izvršavanju programa tek za nekoliko godina, npr. nakon što se promjeni vrsta ili verzija prevodioca (engl. *compiler*). Ovakve je greške najteže otkriti jer se javljaju iznenada i bez jasnog uzroka – zbog neke drugačije tvorničke postavke neke od rijetko korištenih postavki prevodioca ili zbog drugačijeg ponašanja u slučaju nedefiniranih vrijednosti, program koji je godinama ispravno radio odjednom uzrokuje greške. Svaka otkrivena greška nedefiniranog ponašanja uvelike će uštedjeti vrijeme i živce programerima u budućnosti.

- **Otkrivanje zatipaka (tipfelera).** Mnoge pogreške uzrokovane su tipfelerima ili pogreškama u kopiranju i lijepljenju teksta. Vrlo često se u programiranju ponavljaju gotovo identične linije kôda i prirodno je očekivati da će programer pribjeći metodi „kopiraj i zalijepi te zatim ispravi“ umjesto da ispočetka utipkava cijeli izraz. No, trenutak nepažnje u ovakvim slučajevima može rezultirati linijama kôda koje su sintaksom ispravne, ali ipak sadrže neželjene vrijednosti i uzrokuju pogreške. Ispravljanje ovakvih pogrešaka često je vrlo trivijalno te je time veća šteta potrošiti nekoliko sati rada programera za otkrivanje i ispravljanje nečega što bi automatizirani alati mogli obaviti umjesto njega.

- **Dobra skalabilnost.** Kao i većina automatiziranih alata, statička analiza kôda dobro se i lako prilagođava rastu količine izvornog kôda koji treba provjeriti. Alati su primjenjivi neovisno o vrsti i količini kôda, a pokretati se mogu vrlo često, pružajući svakodnevnu kontrolu kôda.

- **Velika učinkovitost u otkrivanju čestih pogrešaka.** Alati za statičku analizu programskog kôda izuzetno su učinkoviti u pronalaženju nekih od najčešćih

pogrešaka kao što su pogreška prelijevanja međuspremnika (engl. *buffer overflow*), ubacivanje SQL upita, nevaljane reference na pokazivače i neinicijalizirane varijable. Primjerice, pri traženju pogrešaka koje omogućuju ubacivanje SQL upita, alati će pomno pregledati i analizirati svaku funkciju i dio programa koji sastavlja SQL upite i ako otkriju da se upit sastavlja na nesiguran način, programera će izvijestiti o otkrivenoj pogrešci i točnom mjestu u programskom kôdu gdje se ona nalazi, što omogućuje brzo i učinkovito otklanjanje pogreške.

2.2 Nedostatci statičke analize kôda

Primarni nedostatak statičke analize kôda krije se u njenoj samoj naravi. Provjerom izvornog kôda bez njegovog izvršavanja nije uvijek moguće otkriti greške uzrokovane međusobnom interakcijom različitih dijelova programa ili interakcijom programa i operacijskog sustava. Problemi istovremenog izvršavanja kôda (engl. *concurrency*) u višedretvenoj okolini jednostavno se neće pokazati statičkom analizom kôda koja ne može predvidjeti redoslijed stvarnog izvršavanja paralelnih operacija. Jednako tako, greške „curenja" memorije je, osim u najosnovnijem obliku, vrlo teško otkriti statičkom analizom kôda. Ispitivanje upravljanja memorijom u pravilu zahtjeva izvršavanje dijelova programa, uz znatne zahtjeve za resursima računala te je primjerenije alatima za dinamičku analizu kôda ili specijaliziranim alatima za analizu radne memorije.

Među ostale nedostatke statičke analize kôda možemo ubrojiti:

- **Pogrešno otkrivanje ranjivosti (engl. *false positives*).** Alati za statičku analizu kôda mogu prijaviti greške i tamo gdje one ne postoje. Lako je moguće da će neobični dijelovi izvornog kôda, pogotovo u situacijama u kojima alat ne može biti siguran u integritet podataka koji prolaze kroz različite dijelove programa, biti proglašeni greškama i tek će naknadni pregled programera utvrditi radi li se doista o grešci ili o lažno pozitivnom rezultatu.

- **Ograničenost u vrstama pogrešaka.** Neke vrste pogrešaka teško je otkriti alatima za statičku analizu kôda. Ovdje posebice ubrajamo razne sigurnosne propuste kao što su problemi autentifikacije, upravljanje pravima pristupa i nesigurna uporaba kriptografije. Iako alati napreduju u podršci otkrivanju ovakvih pogrešaka na razini cijele aplikacije, postotak sigurnosnih propusta koje je moguće otkriti je još uvijek nizak. Još jedna skupina pogrešaka koje alati za statičku analizu kôda ne mogu otkriti su pogreške konfiguracije i pogreške neispravnih podataka. Ove greške uzrokovane su neispravnim vrijednostima koje se nalaze izvan izvornog kôda i razumljivo je da ih automatizirani alati ne mogu otkriti.

- **Poteškoće u radu s kôdom koji se ne može prevesti (engl. *compile*).** Mnogi od automatiziranih alata za statičku analizu programskog kôda imaju poteškoća ili uopće ne mogu analizirati programski kôd ako ga se ne može ispravno prevesti, tj. ako u okolini nije dostupno sve što je potrebno za prevođenje programa. Međutim, u svakodnevnom radu razvijanja softvera, programeri su često u situaciji da izvorni kôd jednostavno nije potpun bilo zato što ovisi o vanjskim bibliotekama i dodacima čiji izvorni kôd trenutno nije dostupan, bilo zato što razvoj programa jednostavno još nije završen do odgovarajuće mjere.

- **Ograničenost programskih jezika.** Iako postoje alati za statičku analizu kôda koji podržavaju nekoliko programskih jezika, većina je alata ipak pisana samo za jedan programski jezik. Tvrtke koje razvijaju softver koristeći se s više jezika ili zbog potreba projekta mijenjaju programski jezik susrest će se s dodatnim troškovima nabavke novih alata i njihovim uhodavanjem u rad.

- **Nerazumijevanje poslovne logike.** Alati za statičku analizu izvornog kôda ne razumiju prave namjere programera i ne mogu otkriti pogreške u poslovnoj logici programa. Ako npr. funkcija za prebacivanje novaca u banci prihvaća negativne iznose, tako da jedan korisnik može prebaciti -100kn (ili bilo koji drugi negativni iznos) drugom korisniku i tako mu zapravo uzeti 100kn, alati obično neće prijaviti pogrešku jer ne razumiju da je slanje negativnog iznosa novca pogrešno u tom kontekstu.

3 Zaključak

Računala i računalni sustavi nezamjenjivi su dio poslovanja svake tvrtke što znači da briga za sigurnost softvera mora biti visoko na popisu prioriteta.

Kako bi se izbjegle pogreške u programskom kôdu i smanjila izloženost ranjivostima, svaki je softver neophodno provjeriti i aktivno raditi na njegovoj sigurnosti. Pri tome je moguće služiti se mnogobrojnim metodama, od statičke i dinamičke analize kôda do penetracijskog testiranja. Niti jedna od metoda upravljanja računalnom sigurnošću ne može se smatrati najboljom ili dostatnom. Svaka ima svoje prednosti i nedostatke i svaka postiže tek dio ukupnog cilja. Tek će sveobuhvatni pristup dati željene rezultate i doista umanjiti rizik za poslovanje.

Uvažavajući nedostatke i primjenjujući svaku tehniku na adekvatan način, postižu se ipak značajni rezultati. Upravljanje računalnom sigurnošću složen je posao koji tek objedinjavanjem višestrukih tehnika i pristupa može polučiti željene rezultate.

Jedna od vrijednih i važnih tehnika je i automatizirana statička analiza izvornog kôda. Uporabom automatiziranih alata ova tehnika postiže izvrstan kompromis između brzine i preciznosti provjeravajući velike količine izvornog kôda u relativno kratkom vremenu.

Prednost automatizirane statičke analize koda je što je skalabilna i jeftina te se može provoditi dnevno, tijekom ranog razvoja, čime sprečava kasnije velike troškove i štedi dragocjeno vrijeme programera. Unatoč njenim nedostacima, statička analiza izvornog kôda u konačnici ipak bitno pridonosi procesu sigurnog i ispravnog razvoja softvera.

4 Literatura

1. **Krebs, Brian.** Panerabread.com Leaks Millions of Customer Records. *Krebs on Security.* [Mrežno] 18. travnja 2019. [Citirano: 26. studenog 2019.] https://krebsonsecurity.com/2018/04/panerabread-com-leaks-millions-of-customer-records/.
2. **RIPS Technologies.** The Techology Leader In Static Application Security Testing. *RIPS Technologies.* [Mrežno] [Citirano: 27. studenog 2019.] https://demo.ripstech.com/scan/112/177.
3. —. Rapid Code Patching. *RIPS Technologies.* [Mrežno] [Citirano: 27. studenog 2019.] https://www.ripstech.com/product/patch-generation/.
4. **GoSecure.** Modern Static Analysis for .NET. *GoSecure.* [Mrežno] 23. studenog 2016. [Citirano: 26. studenog 2019.] https://www.gosecure.net/blog/2016/11/23/modern-static-analysis-net/.
5. **Dutt, Aditya Siddharth.** Lexical Analyzer in C. *Planet Source Code.* [Mrežno] 1. listopada 2015. [Citirano: 27. studenog 2019.] https://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=13124&lngWId=3.
6. **McConnell, Steve.** *Code Complete 2nd Edition.* Washington : Microsoft Press, 2004. ISBN 0-7356-1967-0.