UDC: 004.738.5 INFO M: str. 9-14

DETEKCIJA SIGURNOSNIH PROPUSTA FAZ TESTIRANJEM
DETECTING SECURITY VULNERABILITIES WITH FUZZ TESTING

Aleksandar Nikolić, Goran Sladić, Branko Milosavljević, Zora Konjović
{anikolic,sladicg,mbranko,ftn_zora}@uns.ac.rs
Fakultet Tehničkih Nauka, Univerzitet u Novom Sadu

REZIME: Faz testiranje je veoma efektivna tehnika za detektovanje sigurnosnih propusta u softverskim sistemima. Sastoji se iz prosleđivanja neočekivanih, nevalidnih ili delimično nasumičnih ulaza testiranoj aplikaciji što može dovesti do neočekivanog ponašanja. Fokus faz testiranja u memoriji je modifikacija podataka koji se već nalaze u radnoj memoriji aplikacije kako bi se postiglo faz testiranje. Ovakav pristup ne zahteva prethodno znanje o testiranom sistemu i dobro je prilagođen testiranju sistema zatvorenog koda. Negativne strane ovog pristupa su relativno visok nivo false positive rezultata. U ovom radu predložen je sistem za faz testiranje u memoriji koji koristi snimanje traga izvršenja procesa i analizu propagacije podataka kako bi se smanjio broj false positive rezultata i olakšala analiza test slučajeva.

KLJUČNE REČI: faz testiranje softvera, sigurnost, sigurnosni propusti

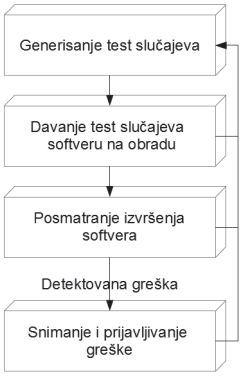
ABSTRACT: Fuzz testing is a very effective technique for software vulnerability detection. It consists of supplying the target application invalid, unexpected or semi-random inputs which may lead to unexpected behavior. The focus of in-memory fuzzing is modification of data already present in the process memory to achieve fuzz testing. This approach requires no prior knowledge about the application and is best suited for testing closed source or obfuscated applications. Major drawback of this approach is a relatively high false positive rate. We propose a system for in-memory fuzz testing which utilizes offline process tracing and data taint analysis to reduce the number of false positive results and improve test case analysis.

KEY WORDS: Fuzzing, software security, software vulnerabilities

1. UVOD

Testiranje predstavlja integralni deo procesa razvoja softvera. Motivi testiranja softvera su, između ostalog, potvrda da implementacija odgovara specifikaciji, potvrda robusnosti i otklanjanje programskih grešaka. Pored ispravnog funkcionisanja softvera neophodno je obezbediti i određeni stepen sigurnosti korisnika i sistema koji softverski proizvod koriste. Otklanjanje sigurnosnih propusta u softveru putem testiranja i u toku razvoja softvera stoga postaje imeprativ. Sigurnosni propust ili ranjivost se definiše kao nedostatak u dizajnu ili implementaciji softverskog sistema koji može da utiče da softver radi suprotno specifikaciji, koji može biti zloupotrebljen radi narušavanja sopstvene bezbednosne politike ili može omogućiti napadaču prisvajanje prava ili kontrole nad sistemom. Ključni deo zaštite informacionog sistema jeste sprečavanje nedozvoljene elevacije privilegija, odnostno zaobilaženje sistema kontrole pristupa [1]. Jedna od metoda za pronalaženje sigurnosnih priopusta jeste faz testiranje.

Faz testiranje ili negativno testiranje predstavlja efikasnu metodu za otkrivanje sigurnosnih propusta, posebno u slučajevima aplikacija zatvorenog koda [2]. Ovaj način testiranja predstavlja automatizovano ili delimično automatizovano testiranje brojnih graničnih slučajeva u softveru, pri čemu se kao ulazni podaci odabiraju nasumični, delimično nasumični ili delimično nevalidni podaci. Sistem koji vrši faz testiranje se naziva fazer (*fuzzer*). Usled visokog stepena automatizacije, ovaj vid testiranja ne zahteva mnogo ekspertskog vremena. Faz testiranje se sastoji od generisanja test slučajeva ulaznih podataka i praćenja izvršenja testiranog softvera u toku obrade. Koraci pri faz testiranju su prikazani na slici 1.



Slika 1. – *Koraci pri faz testiranju.*

Faz testiranjem se obično detektuju programske greške koje dovode do korumpiranja struktura u memoriji. Najrasprostranjeniji oblik ove vrste grešaka je prelivanje bafera.

Postoji više različitih pristupa faz testiranju, a najzančajnija su: faz testiranje bazirano na mutaciji i faz testiranje zasnovano na generisanju. Mutaciono faz testiranja se bazira na mutaciji poznatih i validnih ulaza, a faz testiranje generisanjem se bazira na generisanju novih ulaza na osnovu modela validnih ulaza. Najprostiji oblik mutacije ulaza jeste takozvani bit-flipping [3] kod kojeg se nasumični bajtovi ulaza nasumično menjaju. Iako mutacioni faz sistemi zahtevaju manje znanja o konkretnoj aplikaciji koja se testira i lakši su za implementaciju, obično imaju slabu pokrivenost testiranog koda usled odbijanja mutiranih ulaza kao nevalidnih. Do ovoga može da dođe usled raznih polja sa kontrolnim sumama, pravilima za enkodovanje i logičkim proverama kojih faz sistem nije svestan jer ne poseduje model podataka nad kojim sistem operiše. Sa druge strane, generativni faz sistemi obično imaju

bolju pokrivenost testiranog koda, ali se takođe mora napraviti verni i detaljan model ulaznih podataka. Ovaj model bi uključivao sva znanja o kontrolnim sumama i logičkim proverama koje aplikacija vrši kako bi sistem mogao da generiše ulazne test podatke koji ne bi bili odbačeni kao nevalidni u ranim fazama obrade. Pravljenje detaljnog modela ulaza može da predstalja lak zadatak kada je specifikacija ulaza dostupna, ali i veoma težak zadatak u slučaju vlasničkih formata, obfusciranih i zaštićenih aplikacija. U drugom slučaju neophodno je koristiti tehnike reverznog inžinjeringa kako bi se došlo do do modela ulaza. Za razliku od mutacionih faz testera koji se mogu primenjivati na širok spektar različitih aplikacija bez prevelike modifikacije, generativni faz testeri su primenljivi isključivo na relativno slične aplikacije.

Jedan pristup rešenju ova dva problema: slabe pokrivenosti koda i potrebe za poznavanjem detalja podataka, jeste faz testiranje u memoriji [4]. Umesto generisanja novih ili mutacije postojećih podataka, faz testiranje u memoriji se fokusira na mutaciju podataka koji se već nalaze u memoriji procesa testirane aplikacije. Ovakav pristup prevazilazi probleme kompleksnog generisanja ulaznih podataka čiji detalji nisu poznati i omogućava visok stepen ponovne iskoristivosti sistema na velikom broju aplikacija. Sa druge strane, usled veoma intruzivne prirode, faz testiranje u memoriji ima relativno visok nivo *false positive* rezultata, što u kombinaciji sa nedostatkom konkretnog test slučaja koji bi reprodukovao rezultat zahteva ekspersku analizu rezultata kako bi se otkrili validni [5]. Predloženi pristup ovom problemu ima za cilj smanjenje broja false positive rezultata i olakšavanje analize test slučajeva.

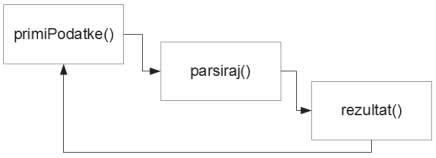
2. FAZ TESTIRANJE U MEMORIJI

Prethodno pominjane metode faz testiranja, generativno ili mutaciono faz testiranje, se fokusiraju na dostavljanju testnih podataka aplikaciji koja se testira, bio to fajl format ili mrežni protokol. U mnogim slučajevima, testiranje samo određenog dela aplikacije je od interesa, ali ovakvim pristupima nemamo mnogo uticaja na to koji se deo koda testira. Cilj faz testiranja u memoriji jeste da se korak mutacije podataka, umesto izvršavanja pre dostavljanja podataka aplikaciji, vrši direktno u memoriji aplikacije, kada su podaci već primljeni na obradu. Ovakav pristup, između ostalog, zaobilazi i probleme komplikovanog ili kompleksnog dela generisanja ulaza, čiji detalji često mogu biti nepoznati. Npr, sloj za šifrovanje ili kodiranje podataka može predstavljati problem kod klasičnih pristupa jer zahteva poznavanje i implementaciju istih mehanizama šifrovanja/kodiranja u sistemu za testiranje, što kod faz testiranja u memoriji nije problem jer se podaci već nalaze u memoriji procesa već dešifrovanja/dekodirani. Mogućnost za precizan odabir podataka koji će biti mutirani i u kom trenutku u toku izvršenja omogućava i precizan odabir delova koda koji se želi testirati čime se postiže bolja i veća pokrivenost koda.

Proces faz testiranja u memoriji se grubo može podeliti na sledeće korake:

1. Odabir dela koda koji će biti testiran
2. Izvršavanje softvera do odabranog dela koda
3. Izmena podataka
4. Izvršenje testiranog dela koda
5. Detektovanje grešaka

Faz testiranje u memoriji se obično sprovodi pomoću *debugger*-a. Kao primer posmatra se graf toka programa sa slike 2.



Slika 2. – *Primer grafa toka programa.*

Ako se želi tesitrati funkcija *parsiraj(), n*a početku funkcije postavlja se *breakpoint* čime se izvršavanje pauzira i kontrola prepušta *debugger*-u. Kroz *debugger* se menjaju testni podaci u memoriji, koji se ili mutiraju slično mutaciji kod *blackbox* faz testiranja, ili se alocira novi deo memorije koji se puni test podacima, a zatim se zamene vrednosti pokazivača da pokazuju na novi deo memorije. Potom se nastavlja izvršenje programa sa izmenjenim podacima do drugog *breakpoint*-a na kraju funckije *parsiraj()*. Kako se programski kod čitavo vreme izvršava unutar *debugger*-a, otkazi i greške mogu biti zabeleženi. Nakon izvršenog testa, potrebno je izvršenje programskog koda vratiti na početak testirane funkcije radi izvršenja sledećeg testa. Ovaj povratak na početak koda koji se testira se može implementirati pomoću snimanja stanja procesa. Po izvršenju instrukcije na koju je postavljen prvi *breakpoint* čuva se kompletno stanje procesa, što uključuje trenutno stanje svih registara i kompletnu memoriju procesa. Nakon što je stanje sačuvano, podaci se mutiraju u memoriji i nastavlja se izvršenje. Po izvršenju naredbe na koju je postavljen drugi *breakpoint*, u slučaju da nije došlo do otkaza, prethodno snimljeno stanje se učitava, čime se, između ostalog, vraća i snimljena verzija programskog brojača. Iznova se menjaju podaci u memoriji i testiranje se nastavlja. U slučaju detektovanja otkaza, učitava se prethodno snimljeno stanje, bez potrebe za ponovnim izvršenjem čitavog procesa.

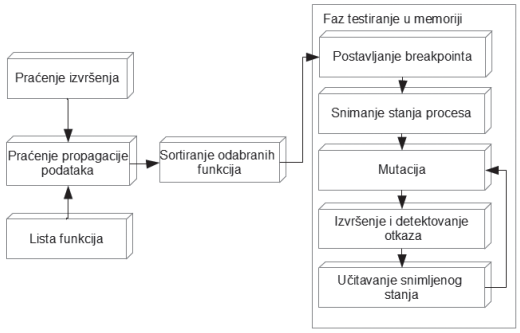
Zbog osobina faz testiranja u memoriji, u slučaju detektovanja otkaza dostupne su samo informacije o tome gde je došlo do otkaza i koji su podaci izmenjeni, a ne i konkretni ulazni podaci koji dovode do otkaza. Ovo predstavlja veliki problem jer dovodi do velikog broja tzv. *false positive* rezultata (test slučajeva za koje se pogrešno smatra da dovode do greške ili otkaza). Svaki slučaj se mora ispitati zasebno da bi se utvrdilo da li se do spornog dela koda uopšte može doći i da bi se došlo do konkretnog ulaza koji dovodi do otkaza. Često ovo nije slučaj jer se može desiti da testirana funkcija uopšte ne zavisi od spoljnih podataka pa potencijalni napadač nad njom nema nikakvu kontrolu, ili pre poziva testirane funkcije mogu postojati provere ili uslovna grananja koja ne bi dozvolila da u normalnom izvršenju proces dođe u stanje detektovanog otkaza. Ovaj problem se delimično može otkloniti korišćenjem praćenja propagacije podataka i testiranja samo onih delova koda do kojih ulazni podaci propagiraju [5].

3. FAZ TESTIRANJE I PRAĆENJE IZVRŠENJA

Praćenjem propagacije ulaznih podataka dobija se egzaktnija slika delova softvera na koje ulazni podaci imaju uticaj. Faz testiranjem tih delova izvršnog koda softvera testira se upravo kod nad kojim ulazni podaci imaju uticaj. Takođe, na

osnovu snimljenog traga izvršenja olakšana je rekonstrukcija ulaznih podataka koji bi doveli do ispoljavanja propusta.

Korišćena je tehnika praćenja propagacije unazad (*backward data tainting*) [6] koja se zasniva na snimanju traga izvršenja programa, a zatim pretraživanja snimljenog traga radi pronalaženja informacija o propagaciji podataka. Kako praćenje propagacije podataka višestruko usporava izvršenje softvera, snimanjem traga izvršenja, a zatim faz testiranjem softvera, koji se izvršava bez usporenja, na osnovu analize traga, postiže se ubrzanje procesa testiranja.Slika 3. prikazuje pojedinačne delove sistema koji su detaljnije opisani u nastavku.



Slika 3. – *Moduli sistema za faz testiranje u memoriji*

Trag izvršenja snimljen tokom praćenja izvršenja procesa se kobinuje sa listom funkcija dobijenom statičkom analizom izvršnog fajla. Na osnovu ovih informacija i praćenja propagacije podataka dobijaju se funkcije čiji parametri zavise od ulaznih podataka i te funkcije postaju kandidati za faz testiranje. Funkcije zatim prolaze evaluaciju koja daje ocenu funkcije na osnovu kompleksnosti i verovatnoće da sadrži greške. Funkcije sa visokom ocenom bivaju testirane prve.

3.1. *Praćenje izvršenja*

Praćenje izvršenja predstavlja poseban oblik izvršenja procesa unutar *debugger*-a. Proces se izvršava instrukciju po instrukciju što omogućava snimanje izvršenih instrukcija, stanja korišćenih registara, adrese pristupa memoriji i vrednosti podataka u memori. Za razliku od statičkog pogleda na izvršni kod, snimljeni trag izvršenja pruža dinamičke informacije: koje putanje kod grananja je proces izvršio, koji su bili sadržaji registara ili memorije i slično. Kako je na osnovu snimljenog traga izvršenja formirana potpuna slika rada procesa, može se ispratiti zavisnost podataka u određenom trenutku izvršenja unazad što omogućava praćenje propagacije ulaznih podataka. Praćenje izvršenja se može implementirati unutar emulatora procesora ili tako što nakon svake instrukcije *debugger* preuzima kontrolu nad programom (*singlesteping*), snima potrebne informacije i nastavlja izvršenje. Oba pristupa višestruko degradiraju brzinu izvršenja procesa te je cilj da praćenje izvršenja procesa izvrši samo jednom, a da se rezultati koriste više puta. U implementaciji predloženog modela korišćen je drugi pristup.

3.2. *Funkcije*

Najniži gradivni blok izvršnog koda, osim samih instrukcija, je osnovni blok, tj. deo izvršnog koda bez skokova i grananja. Pretpostavka je da je osnovni blok isuviše mali deo koda da bi se u testiranju ispoljila stvarna greška, te je kao osnovna jedinica faz testiranja u ovom sistemu uzeta čitava funkcija. U ovom radu ne pravi se razlika između pojmova procedura kod procedurnog programiranja i metoda članica kod objekno orjentisanog programiranja jer sa stanivišta izvršnog, asemblerskog, koda nema bitnih razlika, te obe vrste nazivamo funkcijama. Na nivou asemblerskog koda funkcije je mogće pronaći statičkom analizom izvršnog fajla čiji se kod testira. Funkcije uglavnom imaju specifičan prolog i epilog, a i obično postoje reference na njih u obliku poziva.

3.3. *Propagacija podataka*

Analiza toka programa praćenjem propagacije podataka (*data taint analysis*) ima za cilj definisanje uticaja spoljnih podataka na izvršenje programa koji se posmatra. Kako bi smo pratili propagaciju podataka neophodno je označiti ulazne podatke i pratiti njihov dalji uticaj. Kada se označeni podatak koristi tako da njegova vrednost utiče na rezultat operacije (matematičke ili operacije čitanja/upisivanja u memoriju i slično), odredište rezultata postaje označeni podatak koji se dalje prati. Ovo se naziva propagacija označavanja i predstavlja tranzitivnu relaciju. Korišćenjem informacija prikupljenih tokom praćenja izvršenja moguće je rekonstruisati propagaciju podataka, te analizu propagacije vršiti nakon izvršenja programa. Ovakav pristup se naziva praćenje propagacije unazad (*backward tainting*). Kako je kod faz testiranja u memoriji obično od interesa označenost samo ograničenog broja podataka, ovakav pristup je prirodniji jer se prati samo propagacija podataka od interesa. Kod propagacije podataka unazad, odabira se podatak od interesa i prati se njegovo poreklo unazad kroz snimljeni trag izvršenja. Pretraga snimljenog traga izvršenja se vrši BFS algoritmom dok se ne nađe veza između željenog podatka i prethodno označenog ulaza ili kada više nema putanja za pretragu. Ako je rezultat pretrage pozitivan, dobija se zapis instrukcija koje povezuju zavisnost podatka od interesa od ulaznih podataka. Ovaj zapis instrukcija će biti od velike koristi u procesu analize test slučajeva koji su doveli do otkaza.

U predloženom modelu sistema za faz testiranje u memoriji propagacija podataka unazad se koristi radi pronalaženja funkcija čiji parametri zavise od ulaznih podataka što ih čini validnim kandidatima za faz testiranje. Za svaku funkciju za čiji je bar jedan parametar praćenjem propagacije utvrđeno da zavisi od ulaznih podataka beleže se informacije neophodne za faz testiranje u memoriji. Na osnovu analize propagacije podataka unazad dobija se lista funkcija i njihovih parametara koje treba podvrgnuti faz testiranju jer imaju zavisnosti od ulaznih podataka.

Funkcije dobijene na osnovu analize propagacije podataka treba sortirati prema verovatnoći da sadrže greške kako bi faz testiranje bilo usmereno ka boljim kandidatima. Jedan od parametara verovatnoće da funkcija sadrži grešku je njena kompleksnost. Što kompleksnija funkcija, veća je verovatnoća za grešku. U ovom modelu kompleksnost funkcije merimo brojem baznih blokova od kojih se funkcija sastoji. Kompleksnosti funkcije doprinose i pozivi drugih funkcija pa i taj podatak ulazi u ocenu funkcije. Ako evaluirana funkcija poziva funkcije za koje je poznato da dovode do sigurnosnih propusta (kao što su *strcpy*, *memcpy*, *sprintf* i slične), dodatno se povećava ocena. Najzad, i za pojedine instrukcije je pozna-

to da dovode do sigurnosnih propusta (obično instrukcije sa *rep* prefiksom za operacije nad stringovima) pa i one ulaze u ocenu. Svaki činilac je ponderisan vrednostima dobijenim na osnovu genetskog algoritma, iz istraživanja datog u [7].

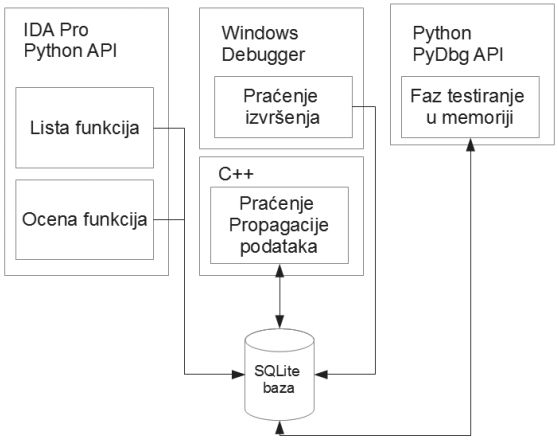
3.4. *Faz testiranje u memoriji*

Nakon sortiranja funkcija, bira se prva i prelazi se na faz testiranje. Na osnovu podataka o parametrima funkcije, isti bivaju mutirani u memoriji. Prati se izvršenje funkcije do povratka u prethodnu, nakon čega se učitava sačuvano stanje procesa i nastavlja testiranje. Jednostavnom analizom sadržaja razlikuju se tri vrste parametara: neposredna vrednost, pokazivač na deo memorije i pokazivač na ASCII string. U slučaju neposredne vrednosti, parametar se smatra celobrojnom vrednošću i mutira na odgovarajući način. U slučaju pokazivača na ASCII string, parametar se mutira tako što se string produžava, briše se terminirajući NULL karakter, dodaju se specijalni znakovi formata i slično. Deo memorije na hipu se mutira *bitflipping* metodom. Kada se iskoriste sve kombinacije faz testiranja jedne funkcije, prelazi se na sledeću.

Nakon faz testiranja, ekspert analizira zabeležene informacije radi procene validnosti sigurnosnog propusta.

4. IMPLEMENTACIJA MODELA

Model sistema za faz testiranje u memoriji je implementiran na Windows operativnom sistemu sa 32 bitnom Intel X86 arhitekturom. Rasprostranjenost softvera zatvorenog koda, gde faz testiranje ima više smisla, je daleko veća na Windows sistemima u odnosu na Unix/Linux familiju operativnih sistema. Predložene metode su uopštene i mogu se implementirati i na drugim platformama. Na slici 4. prikazane su tehnologije korišćene pri implementaciji.



Slika 4. – *Arhitektura implementiranog sistema.*

4.1. *Implementacija praćenja izvršenja*

Praćenje izvršenja je jedan od ključnih delova predloženog sistema. Kao osnova preuzet je *Visual Data Tracer* [8] koji predstavlja proširenje za *Windows Debugger*, standardni *debugger* na Windows operativnim sistemima. Proces čije se izvršenje prati se izvršava instrukciju po instrukciju, tj. proces je u *singlestep* režimu rada. Pre izvršenja svake instrukcije beleže se informacije koje će biti neophodne pri analizi propagacije podataka. Za svaku podržanu instrukciju se beleže sledeće informacije:

1. Menmonik instrukcije
2. Odredišni operand
3. Izvorni operand
4. Zavisnosti izvornog operanda – u slučaju pristupa memorijskoj lokaciji
5. Pokazivač – memorijska adresa kojoj se pristupa
6. Tekstualni prikaz – *disassemble*-ovani prikaz cele instrukcije

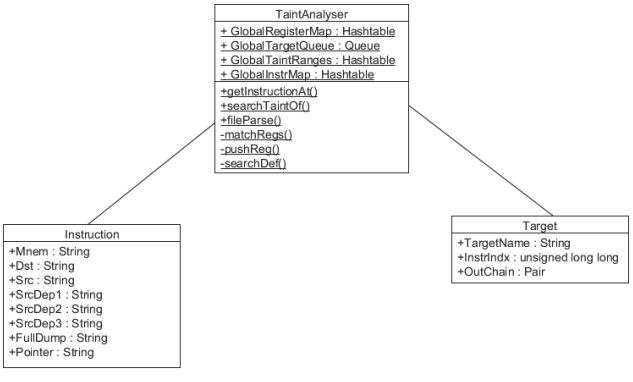
Postoje instrukcije čije izvršenje implicitno menja vrednosti određenih registara pa je i njih potrebno posebno obraditi. To su RDTSC, RDPMC i CPUID instrukcije koje nemaju ulazne operande, ali njihovo izvršenje prepisuje sadržaje EAX, EDX, EBX i ECX registara, što će biti bitno pri analizi propagacije podataka [12].

Instrukcije koje za posledicu imaju promenu lokacije izvršenja koda, kao što su CALL, RET, SYSENTER, INT i slične, nisu direktno od interesa pri analizi propagacije podataka jer ni na jedan način ne utiču na podatke, ali su neophodne za utvrđivanje poziva funkcija i praćenje toka izvršenja programa [13]. U slučaju ovih instrukcija zapisuje se samo tekstualni prikaz cele instrukcije.

Praćenje izvršenja se vrši tako što se program izvršava unutar debugger-a do trenutka kada se ulazni podaci nalaze u memoriji, bilo pročitani iz fajla ili sa mreže. Potrebno je ručno zabeležiti adrese ulaznih podataka u memoriji koje će biti korišćene u toku analize propagacije podataka. Nakon toga se učitava proširenje za praćenje izvršenja koje započinje dalje izvršenje programa i pre svake izvršene instrukcije beleži potrebne informacije. Praćenje se vrši do prekida izvršenja programa, bilo krajem izvršenja ili zaustavljanjem u debugger-u, a sačuvane informacije se zapisuju u fajl na disku, a zatim u bazu podataka.

4.2. *Analiza propagacije podataka*

Analizi propagacije podataka se pristupa nakon snimanja traga izvršenja. Na slici 5. prikazan je klasni dijagram sistema za analizu propagacije podataka. *TaintAnalyser* parsira navedeni fajl u kom se nalazi snimljeni trag izvršenja, svaki unos u fajlu predstavlja jednu instrukciju.



Slika 5. – *UML dijagram klasa modula za analizu propagacije podataka*

Svaka parsirana instrukcija se smešta u globalnu heš tabelu instrukcija *GlobalInstrMap*, redni broj instrukcije služi kao ključ. Postavljaju se opsezi memorije za koje se smatra da sadrže ulazne podatke. Klasa *Target* predstavlja podatak za koji se traži veza sa ulaznim podacima. Atribut *TargetName* je tekstualna oznaka podatka što može biti registar ili memorijska lokacija. Indeks instrukcije u klasi *Target* označava od koje instrukcije treba početi praćenje propagacije u nazad.

Nakon što je odabran podatak koji se analizira, u slučaju memorijske adrese, proverava se da li ona upada u neki od opsega označenih da sadrže ulazne podatke. Ako je ova provera pozitivna, potraga se završava i podatak je u direktnoj vezi sa ulazom. Ako to nije slučaj pretražuje se lista instrukcija za mestom gde je podatak definisan, odnostno traži se instrukcija u snimljenom tragu izvršenja gde je traženi podatak bio odredišni operand. Kada se nađe instrukcija gde je podatak definisan, proverava se da li njen izvorni operand upada u neki od definisanih opsega. Ako to nije slučaj, svaki deo izvornog operanda pronađene instrukcije se dodaje u listu kao nova instanca klase *Target*. Pretraga se nastavlja dok god ima elemenata u listi. Ako tokom pretrage nijedan operand ne upada u neki od opsega, zaključuje se da početni podatak ne zavisi od ulaznih podataka. U implementaciji predloženog sistema za faz testiranje u memoriji, analiza propagacije podataka u nazad se primenjuje na parametre funkcija. Implementirana je podrška za funkcije koje koriste *cdecl* ili *stdcall* konvenciju pozivanja. Kod funkcija koje poštuju ovu konvenciju, parametri se pre poziva funkcije smeštaju na stek i to na dva načina. Instrukcijom PUSH koja sadržaj operanda smešta na vrh steka. Drugi način je korišćenjem MOV instrukcije kojoj je odredišni operand vrh steka. Kako bi se pronašli argumenti funkcije, pretražuju se instrukcije odmah pre poziva funkcije za jednim od dva pomenuta načina smeštanja argumenata. Pretraga za argumentima se terminira kada se naiđe na instrukciju skoka, uslovnog grananja ili RET instrukciju koja označava povratak iz druge funkcije.

Analiza propagacije podataka se vrši za svaku funkciju od interesa i pozitivni rezultati se smeštaju u *SQLite* bazu podataka radi lakšeg pristupa u fazi testiranja. Čuvaju se informacije o propagaciji podataka kao i lokacija argumenta na steku funkcije. Lokacija argumenta se čuva relativno u odnosu na ESP registar, jer se konkretne adrese mogu promeniti u toku testiranja.

4.3. *Odabir funkcija za testiranje*

Pre testiranja neophodno je odabrati izvršni fajl koji se testira. Potom je potrebno formirati listu svih funkcija izvršnog fajla kako bi se u kombinaciji sa analizom propagacije podataka došlo do onih čiji parametri zavise od ulaznih podataka i koje ima smisla testirati. U ove svrhe je iskorišćena slobodna verzija *IDA Pro* [9] alata koja uz upotrebu Python API-a omogućava statičku analizu izvršnih fajlova. Funkcije obično poseduju prolog u kom se alocira memorija na steku za lokalne promenljive i epilog u kom se oslobađa stek (postoje razlike između *cdecl* i *stdcall* funkcija) i vrši povratak u prethodnu funkciju. Instrukcije u prologu i epilogu su slične za većinu funkcija pa je na osnovu toga moguće definisati početak i kraj funkcije.

Nakon analize propagacije podataka, potrebno je sortirati funkcije po proceni verovatnoće da sadrže grešku. Svaka MOV instrukcija čiji jedan operand pristupa memoriji doprinosi oceni za 1. LEA instrukcija doprinosi sa 1.5. CALL instrukcija povećaca ocenu za 3, a u slučaju da se pozvana funkcija nalazi u listi opasnih funkcija, ocena se povećava za 6. Lista opasnih funkcija sadrži *strcpy*, *strncpy*, *memset*, *memcpy*, *sprintf* i *snprintf*. Instrukcije sa prefiksom REP takođe povećavaju ocenu za 3. Kako broj baznih blokova funkcije omogućava procenu kompleksnosti funkcije, isti doprinosti oceni funkcije sa množiocem 0.25. Ocene i množioci su preuzeti iz [7]. Bazni blokovi se identifikuju tako što se odredišna adresa svakog skoka ili uslovnog grananja uzima za početak, a bilo koja instrukcija koja menja tok izvršavanja programa za kraj baznog bloka. Funkcije se sortiraju po oceni i one sa najvišom bivaju testirane prve.

4.4. *Implementacija faz testiranja*

Proces faz testiranja u memoriji je implementiran uz oslonac na *PyDbg* Python [10] biblioteku. Iz baze se čitaju informacije o prvoj funkciji koju treba testirati. Postavljaju se *breakpoint*-i na početak i kraj funkcije. Potom se startuje izvršenje programa i proslede se odgovarajući ulazi. Kada izvršenje stane na prvom breakpointu prvi put, snima se stanje programa.

Kako programi na Windows platformi obično imaju više niti koje se simultano izvršavaju, potrebno je sačuvati kontekst svake pojedinačne niti. Takođe, neophodno je sačuvati i sadrža radne memorije procesa. Virtualni adresni prostor na Windows sistemu svakog procesa je veličine 4 gigabajta. Svaki proces ima na raspolaganju adrese u opsegu od 0x00000000 do 0x7FFFFFFF, više adrese su rezervisane za kernel i ne može im se pristupiti iz običnog procesa direktno. Sve adrese koje pripadaju procesu su podeljene u stranica i svaka stranica ima svoje dozvole kao što su PAGE_READONLY (moguće je isključivo čitanje), PAGE_EXECUTE_READ (moguće je čitanje i izvršavanje koda), PAGE_GUARD (generiše se izuzetak pri prvom pristupu) i PAGE_NOACCESS (zabranjen bilo koji oblik pristupa). Pri snimanju stanja memorije procesa mogu se ignorisati memorijske stranice sa prethodnim dozvolama pošto se one neće menjati tokom izvršenja. Ovi koraci pri snimanju stanja procesa su neophodni zbog ograničene radne memorije koju sistem za faz testiranje ima na raspolaganju.

Sledeći faza koja se sprovodi je mutacija parametara funkcije. Iz analize propagacije podataka dostupne su informacije o lokaciji parametra i potrebno je utvrditi šta taj parametar predstavlja. Parametar može sadržati direktnu vrednost ili memorijsku adresu. Pomoću *PyDbg* API-a može se utvrditi da li se radi o adresi na hipu ili steku ili o direktnoj vrednosti. Takođe, *PyDbg* omogućava menjanje sadržaja adrese što je neophodno za mutiranje. Samo mutiranje se vrši po postupku opisanom ranije. Ako funkcija koja testira ima više parametara koji zavise od ulaza, uvek se mutira bar jedan ili kombinacija.

Nakon mutiranja parametra, nastavlja se izvršavaje procesa. Ako ne dođe do otkaza, trenutno stanje procesa se menja snimljenim tako što se vraćaju snimljeni konteksti niti. U

slučaju otkaza *debugger* prekida izvršenje procesa i beleži informacije o otkazu.

Na osnovu zabeleženih informacija, ekspert pristupa analizi. Prvi cilj je utvrditi da li se i kako, na osnovu snimljenih informacija, mogu direktno izmeniti ulazni podaci koji dovode do istog otkaza. U ovome pomažu rezultati analize propagacije podataka.

5. ZAKLJUČAK

U današnje doba široke rasprostranjenosti Interneta i personalnih računara, sigurnosni propust u široko rasprostranjenim softverskim aplikacijama predstavlja rizik za veliki broj korisnika. Softver zatvorenog koda uvodi dodtanu kompleksnost pri testiranju, jer je za pojedine metode testiranja, kao što je statička naliza, neophodan izvorni kod [11]. Pronalaženje i otklanjanje sigurnosnih propusta je jedan od prioriteta u toku održavanja softverskog proizvoda ali zahtevi tržišta za brzim razvojem postavljaju ograničenja u pogledu vremena dostupnog za testiranje.

Faz testiranje predstavlja vremenski efikasan način pronalaženja programskih grešaka te je stoga jako popularan. Cilj svakog pristupa faz testiranju je da ima što viši stepen automatizacije i da pokriva što veći deo testiranog koda, tj. da daje bolje rezultate. S tim ciljem je razvijena tehnika faz testiranja u memoriji. Korak generisanja ulaznih podataka se preskače jer se vrši mutacija validnih podataka koji se nalaze u radnoj memoriji testiranog procesa. Zbog činjenice da se mutacija podataka vrši direktno u memoriji, u slučaju detektovanog otkaza, ne postoje konkretni ulazni podaci koji dovode do tog otkaza. Veliki broj zabeleženih otkaza predstavlja false positive rezultate koje nije moguće reprodukovati bilo kojim ulaznih podacima. Otkrivanje false positive rezultata se ne može automatizovati te taj teret pada na eksperta koji analizira zabeležene otkaze. Takođe, dodatno vreme od strane eksperta je potrebno za rekonstrukciju konkretnih ulaznih podataka koji dovode do otkaza.

U ovom radu dat je predlog modela testiranja koji obezbeđuje poboljšanje rezultata dobijenih faz testiranjem u memoriji smanjenjem broja false positive rezultata i olakšavanjem postupka rekonstrukcije ulaznih podataka. Smanjenje false positive rezultata se postiže faz testiranjem samo podataka za koje je analizom propagacije podataka utvrđeno da zavise od ulaza. U predloženom rešenju demonstrirana je upotreba offline analize praćenja propagacije podataka u svrhe faz testiranja u memoriji. Predloženi sistem je implementiran za Windows operativni sistem na Intel x86 arhitekturi procesora.

Daljim proširenjem i optimizacijom implementiranog sistema mogao bi da se postigne viši stepen automatizacije i bolji rezultati. Potencijalno ubrzanje i optimizacija procesa praćenja propagacije bi se mogla postići implementacijom alata nezavisnog od samog debugger-a. Presretanjem funkcija operativnog sistema koje rukuju ulazima na niskom nivou, funkcije za čitanje iz fajlova ili za čitanje podataka sa mreže, i čuvanjem informacija o njihovim parametrima i povratnim vrednostima može se postići automatizovano označavanje opsega memorije. Primenjivanjem postupaka simboličkog izvršenja i rešavanja ograničenja, slično metodama predstavljenim u SAGE [12] sistemu, mogla bi se postići preciznija mutacija podataka i filtriranje detektovanih otkaza.

REFERENCE

[1] G. Sladić. Kontekstno zavisna kontrola pristupa. *Info M*, 41, 2012.

[2] A. Takanen, J. DeMott, and C. Miller. Fuzzing for Software Security Testing and Quality Assurance. *Artech House Information Security and Privacy*, 2008.

[3] C. Miller. How Smart is Intelligent Fuzzing. *DEFCON*, 15, 2005.

[4] M. Sutton, Fuzzing: Brute Force Vulnerability Discovery. *Addison-Wesley Professional,* 2007.

[5] V. Iozzo. 0-knowledge fuzzing. *BlackHat DC*, 2010.

[6] J. Auto, Triaging Bugs with Dynamic Dataflow Analysis, *Source Conference*, 2009.

[7] B. Spasojević, Primjena genetskih algoritama u postupku otkrivanja propusta protokola, *Sveučilište u Zagrebu*, 2008.

[8] R. Branco. Dynamic Program Analysis and Software Exploitation. *Internet*: http://phrack.org/issues.html?issue=67&id=10#article

[9] IDA Pro Demo. *Internet*: http://www.hex-rays.com/products/ida/support/download_demo.shtml

[10] P. Ammini, PyDbg. Internet: https://github.com/OpenRCE/pydbg

[11] D. Vuković. STASEC – alat za otkrivanje sigurnosnih propusta web aplikacija statičkom analizom java izvornog koda. *Info M*, 39, 2011.

[12] P. Godefroid, Automated Whitebox Fuzz Testing. *Microsoft Research*, 2008.

[13] Intel® 64 and IA-32 Architectures Software Developer Manuals. *Intel Corporation*,2012.

MSc Aleksandar Nikolić, Fakultet tehničkih nauka, Univerzitet u Novom Sadu.
Kontakt: anikolic@uns.ac.rs
Oblasti interesovanja: bezbednost informacionih sistema, analiza programa, programske ranjivosti

dr Goran Sladić, Fakultet tehničkih nauka, Univerzitet u Novom Sadu.
Kontakt: sladicg@uns.ac.rs
Oblasti interesovanja: bezbednost informacija, upravljanje dokumentima, workflow sistemi, XML tehnologije

prof. dr Branko Milosavljević, Fakultet tehničkih nauka, Univerzitet u Novom Sadu.
Kontakt: mbranko@uns.ac.rs
Oblasti interesovanja: pretraživanje informacija, upravljanje dokumentima, kontrola pristupa, digitalne biblioteke

prof. dr Zora Konjović, Fakultet tehničkih nauka, Univerzitet u Novom Sadu.
Kontakt: ftn_zora@uns.ac.rs
Oblasti interesovanja: veštačka inteligencija, upravljanje dokumentima, digitalne biblioteke, geoinformacioni sistemi