Univerzitet u Beogradu
Matematički fakultet

Seminarski rad iz predmeta Verifikacija softvera

Analiza aplikacije Password manager

Student: Emilija Stevanović 100/2019,
Avgust, 2024

Sadržaj

1 Uvod

U ovom seminarskom radu je izvrešena analiza konzolne i gui aplikacije *Password manager* , koji je otvorenog koda i preuzet je sa github repozitorijuma. Jednostavan za korišćenje menadžer lozinki sa enkripcijom za njihovo sigurno čuvanje.

2 Testovi jedinica koda i pokrivenost koda

Dinamička verifikacija softvera obuhvata tehnike ispitivanja ispravnosti softvera u toku njegovog izvršavanja. Najčešći vid dinamičke verifikacije softvera, a i verifikacije softvera uopšte, je testiranje. Time se smanjuje verovatnoća greške. Potrebno je isplanirati, implementirati i eveluirati rezultate testova.

U ovom radu je izvršeno testiranje jedinica koda pomocu CuTest biblioteke. CuTest je jednostavan framework za jedinicne testove (unit testing) u C jeziku. Dizajniran je tako da bude mali i lako razumljiv, što ga čini idealnim za projekte koji ne žele da se oslanjaju na složenije alate. CuTest je veoma mali, sa samo nekoliko datoteka i linija koda, što ga čini brzim za kompajliranje i jednostavnim za integraciju u već postojeće projekte. Pošto je CuTest napisan u čistom C jeziku, lako se može preneti na različite platforme i okruženje, dakle, portabilan je. Testovi se mogu organizovati u test suite-ove, što omogućava grupisanje srodnih testova i olakšava njihovo pokretanje, a rezultat testova je vidljiv na konzoli.

Testirane su funkcije za enkripciju i dekripciju stringova *test_b64.c* tako što je provereno da li su funkcije za enkripciju i dekripciju zaista inverzne. Takođe, testirane su i randomizovane funkcije *test_rand.c*, da li vraćaju stringove sa dozvoljenim karakterima ili da li je došlo nepredviđene vrednosti za int i slično. Fajl *test_runner.c* pokreće sve implementirane testove u skladu sa CuTest-om. Za svaki fajl koji predstvlja test jedinice koda je napravljena Suite funkcija, koja se dodaje, zatim se svi dodati suit-ovi pokreću i najzad nakon ispisa rezultata memorija oslobađa.

Nakon što smo se uverili da su svi testovi prošli možemo uz pokretanje testova da pratimo pokrivenost koda tim testovima koja je izvršena lcov alatom. LCOV je alat za vizualizaciju pokrivenosti koda (code coverage) u projektima napisanima u C i C++ jezicima. On je frontend za alat gcov, koji je deo GNU Compiler Collection (GCC), i omogućava generisanje detaljnih izveštaja o pokrivenosti koda u HTML formatu. LCOV meri koliko su različiti delovi koda, poput funkcija, linija i grananja, pokriveni testovima. Ovo pomaže programerima da identifikuju delove koda koji nisu testirani i koji potencijalno sadrže greške. Da bi se pokrenuo lcov alat prvo je potrebno kompajlirati izvorni kod koji pokreće testove sa opcijom za peikupljanje informacija o pokrivenosti –coverage . Posle pokretanja testova, prikupljaju se podaci o pokrivenosti pomocu lcov-a lcov –capture –directory . –output-file coverage.info . Napokon, može se generisati izveštaj genhtml coverage.info –output-directory out .

2.1 Analiza izveštaja

Na ocnovu izveštaja koji je lcov generisao vidimo da su funkcije b64_encode i b64_decode pozvane jednom, imamo i informacije u kom fajlu i na kojoj liniji počinje njihova imlementacija. Tako možemo uočiti koje funkcije su pokrivene testovima i koliko puta su pozivane. Sa druge strane, rand_int funkcija je pozvana 1001 put, a kriticna tačka te funkcije je linija 14 koja je izvrašena 2002 puta. Iako je ovo kriptografski program i profajliranje nema mnogo smisla, moglo bi se zaključiti da bi ovo moglo uticati da celokupno vreme izvršavanja. Krajnje statistike koje nam je lcov dao su sledeće :

Ključni podaci

- **Ukupna pokrivenost linija koda:** 76.6%
- **Ukupna pokrivenost funkcija:** 81.4%
- **Izveštaj generisan:** 17. avgusta 2024. godine u 12:15:13

Detalji po direktorijumima

password-manager/source/console

- **Pokrivenost linija:** 98.9% (88/89 linija)
- **Pokrivenost funkcija:** 100% (4/4 funkcija)

unit_tests

- **Pokrivenost linija:** 68.7% (171/249 linija)
- **Pokrivenost funkcija:** 79.5% (31/39 funkcija)

Zaključak: Visoka je pokrivenost u password-manager-master/source/console modulu, a to ukazuje na to da je ovaj deo koda dobro testiran i verovatno stabilan. Niža pokrivenost unit_tests direktorijumu može da ukaže na to da testovi koji su implementirani ne pokrivaju sve moguće scenarije i grane koda koje testiraju.

3 Valgrind

Valgrind je platforma za izgradnju alata za dinamičku analizu. Postoje Valgrind alati koji mogu automatski otkriti mnoge greške u upravljanju memorijom i nitima, i detaljno analizirati programe. Takođe, može se koristiti za pravljenje novih alata. Svi Valgrind alati rade na istoj osnovi, a informacije koje se emituju variraju[1].

$$Jezgro\_Valgrind - a + Alat\_koji\_se\_dodaje = Alat\_Valgrind - a$$

Veliki broj korisnih alata se isporučuje kao standard:

- *Memcheck* je detektor grešaka u memoriji. Posebno značajan za programe napisane u C i C++.
- *Cachegrind* je profajler za rad sa keš memorijom i predviđanje granjanja. Pomaže da programi rade brže.
- *Callgrind* je keš profajler koji generiše graf poziva. Ima izvesno preklapanje sa Cachegrind-om, ali takođe prikuplja neke informacije koje Cachegrind nema.
- *Helgrind* je detektor grešakau radu sa nitima.
- *DRD* je takođe detektor grešaka u radu sa nitima i koristi različite tehnike.
- *Massif* je hip profiler. Cilj je da programi koriste manje memorije.
- *DHAT* je druga vrsta hip profajlera. Pomaže pri razumevanje problema životnog veka bloka i rasporeda.
- *BBV* je generator blok vektora. Korisno je koji za istraživanje i razvoj arhitekture računara.

3.1 Massif

Massif je profajler za hip memoriju. Meri koliko hip memorije program koristi. Ovo uključuje kako upotrebljen prostor, tako i dodatne bajtove alocirane radi vođenja evidencije i postizanja poravnanja. Takođe može meriti veličinu steka programa, iako to podrazumevano ne radi. Postoje određene curenja memorije koja se ne detektuju tradicionalnim alatima za proveru curenja, kao što je Memcheck. To je zato što memorija zapravo nije izgubljena – postoji pokazivač na nju – ali nije u upotrebi. Programi koji imaju curenje kao što je ovo mogu nepotrebno povećavati količinu memorije koju koriste tokom vremena. Massif može pomoći u identifikaciji ovih curenja. Važno je napomenuti da Massif ne govori samo koliko hip memorije vaš program koristi, već pruža i vrlo detaljne informacije koje ukazuju na to koje delove vašeg programa je odgovorno za alociranje hip memorije.

Massif alat se pokreće `valgrind -tool=massif ./program` , a pri pokretanju je moguće uključiti naredne opcije:

- –**heap**: Profilisanje korišćenja gomile (heap). Ovo je podrazumevana opcija.
  Primer: `valgrind -tool=massif -heap=yes ./program`
- –**stacks**: Profilisanje korišćenja steka (stack). Podrazumevana vrednost je `yes`.
  Primer: `valgrind -tool=massif -stacks=yes ./program`
- –**time-unit**: Određuje jedinicu vremena za uzorkovanje. Opcije su:
  - `i` (instrukcije)
  - `ms` (milisekunde)
  - `B` (bajtovi dodeljeni hipu)

  Primer: `valgrind -tool=massif -time-unit=ms ./program`
- –**depth**: Postavlja maksimalni broj stekova koje Massif prati. Podrazumevana vrednost je 30.
  Primer: `valgrind -tool=massif -depth=20 ./program`
- –**threshold**: Postavlja prag (u procentima) ispod kojeg se promene u memorijskom otisku neće evidentirati. Podrazumevana vrednost je 1%.
  Primer: `valgrind -tool=massif -threshold=0.1 ./program`

- –**max-snapshots**: Određuje maksimalni broj snimaka koje Massif pravi tokom izvršavanja programa. Podrazumevana vrednost je 100.
  Primer: `valgrind -tool=massif -max-snapshots=200 ./program`
- –**detailed-freq**: Određuje učestalost detaljnih snimaka, izražena kao broj osnovnih snimaka između dva detaljna snimka. Podrazumevana vrednost je 10.
  Primer: `valgrind -tool=massif -detailed-freq=5 ./program`
- –**alloc-fn**: Definiše specifičnu funkciju za alokaciju memorije koja će biti praćena.
  Primer: `valgrind -tool=massif -alloc-fn=my_malloc ./program`
- –**massif-out-file**: Određuje izlazni fajl u koji će Massif snimiti rezultate.
  Primer: `valgrind -tool=massif -massif-out-file=massif_output.out ./program`

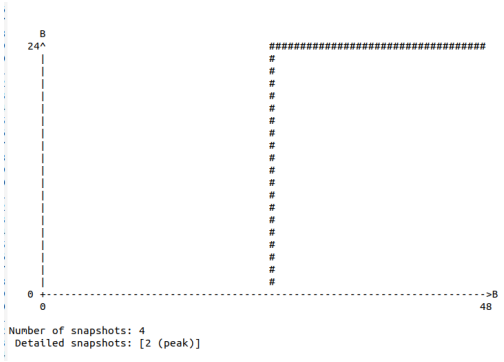
Alat je pokrenut za različite načine korišćenja password-manager aplikacije, u cilju provere korišćenje hip memorije u različitim situacijama (generisanje slučajne lozinke, enkripcija i dekripcija stringova, izlistavanje svih lozinki, pretraživanje lozinki sa wildcard-om ...).

3.1.1 Analiza izlaza

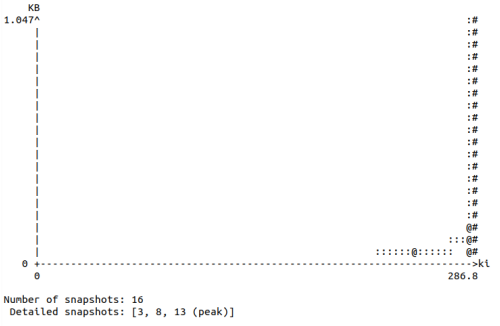
Lako se moze pratiti kako program koristi memoriju na grafikonima koje je generisao ms_print. Snimci koji su normalni su obeleženu ':', detaljni snimci su predtsavljeni na grafikonu tačkama '@'. Konačno snimak koji je dostgao vrhunac korišćenja memorije je označen '#'. Na primer, od interesa je kako radim program dok pretražuje određenu šifru samo na osnovu wildcard-a (Slika 1 f). Razlog zbog čega je početak grafa prazan je to da Massif podrazumevano koristi instraction executed kao jedinicu vremena. Za programe koji se izvršavaju vrlo kratko većina izvršenih instrukcija odnosi se na učitavanje i dinamičko povezivanje programa. Korisniji grafik se može dobiti opcijom `-time-unit=B` za korišćenje bajtova kao jedinice vremena. Broj snapshota koji su zebeleženi od kojih su dva detaljna (3, 8). Podrazumevano, svaki deseti snimak je detaljan, iako se to može promeniti opcijom `-detailed-freq`. Na kraju, postoji najviše jedan snimak vršnog opterećenja (peak snapshot). Snimak koji je dostigao pik i on je detaljan snimak i beleži trenutak kada je potrošnja memorije bila najveća. Ovde je 8 pik. Detalje o detaljno obrađenim snapshot-ovima mogu se pogledai ispod grafika. Primećujemo da potrošnja memorije raste do 8. Za detljne snapshot-ove je data lista funkcija i fajlova koje su alocirale memoriju na hipu, a za

svaki snapshot se za kada je i koliku memorio zauzeo, korisnu (onu koju program koristi) i dodatnu (zauzeta memorija, ali se ne koristi). Kao sto je već naznačeno 8. predstavlja pik: funkcija glob_compile koristi 90.78% hip memorije 10240B. Od 9. do 10. potrošnja opada. Na kraju zaključujemo da kada tražimo šifru pomoću wildcarda program troši najvise 11264B memorije, a na kraju nema neoslobođene memorije. Dakle, nema curenja memorije. To se možemo uveriti i za ostale massif output fajlove.

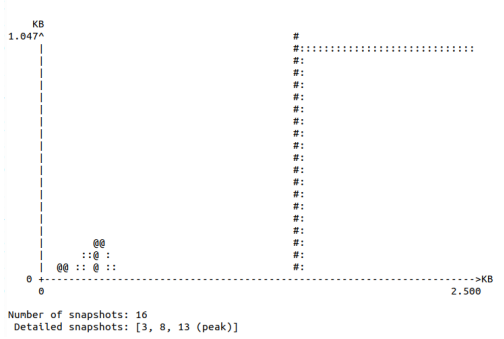
(a) b64 dec



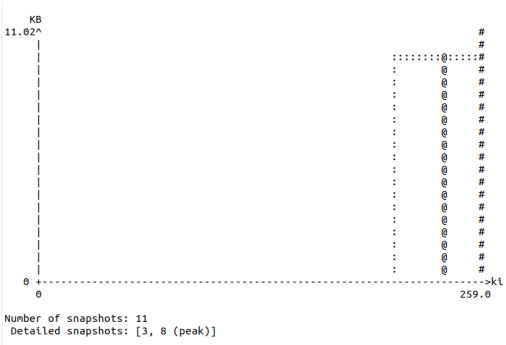
(b) b64 dec bolje



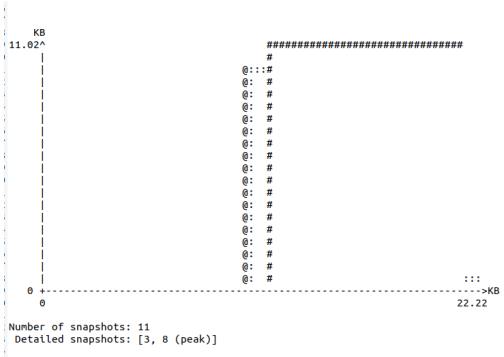
(c) Generisanje



(d) Generisanje bolje



(e) Pretraga



(f) Pretraga bolje

Slika 1: Rezultati

3.2 Callgrind

Callgrind je alat za profajliranje koji beleži istoriju poziva funkcija tokom izvršavanja programa preko call-graph-a (graf poziva). Podaci koji se podrazumevano prikupljaju obuhvataju broj izvršenih instrukcija, njihov odnos prema linijama koda, odnos pozivaoca/pozvane funkcije između funkcija i broj takvih poziva. Opciono, simulacija keša i/ili predviđanje grananja (slično Cachegrind-u) može pružiti dodatne informacije o ponašanju programa tokom izvršavanja.

Podaci profila se zapisuju u datoteku pri završetku programa. Za prikaz podataka i interaktivno upravljanje profajliranjem, koriste se dva alata sa komandne linije:

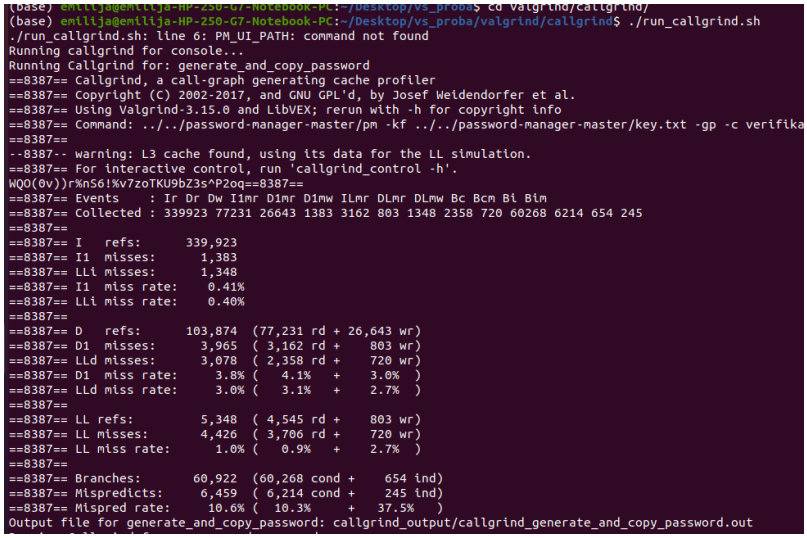
- `callgrind_annotate` - Ova komanda čita podatke profila i ispisuje sortiranu listu funkcija, opciono sa anotacijama izvornog koda.
- `callgrind_control` - Ova komanda vam omogućava interaktivno praćenje i upravljanje statusom programa koji se trenutno izvršava pod kontrolom Callgrind-a, bez zaustavljanja programa. Možete dobiti statističke informacije, trenutni stek poziva, zatražiti resetovanje brojača ili snimanje podataka profila.

Za grafičku vizualizaciju podataka, preporučuje se korišćenje KCachegrind-a, koji je GUI baziran na KDE/Qt i olakšava navigaciju kroz veliku količinu podataka koje Callgrind generiše.

Alat je pokrenut za enkodiranje i dekodiranje stringa, generisanje kluča, pretragu ključeva,ispis istih itd. Uključena je opcija `-cache-sim=yes` koji omogućava simulaciju i detaljnu analizu ponašanja keša tokom izvršenja programa. Po defaultu, samo pristupi za čitanje instrukcija će biti brojani ("Ir"). Sa simulacijom keša, omogućeno je praćenje: promašaji keša za čitanje instrukcija ("I1mr"/"ILmr"), pristupi za čitanje podataka ("Dr") i povezani promašaji keša ("D1mr"/"DLmr"), pristupi za pisanje podataka ("Dw") i povezani promašaji keša ("D1mw"/"DLmw"). Svaki scenario ima poseban .out fajl, koji je kasnije anotiran i za koji je generisan graf poziva u graphs direktorijumu. Callgrind je pratio konzolnu aplikaciju,ali je pokrenut i za gui aplikaciju.

Na primer, razmatra se izlaz za generisanje nove lozinke i njeno kopiranje. Procenat promašaja keša za instrukcije I1 i LLi (poslednji nivo) je 0.41% . Keš za podatke D ima procenat promašaja 3.8%, a pogrešna predikcija grana je 10.6%,

Slika2. Upoređivanjem statistike promašaja keša za druge funkcionalnosti aplikacije zaaključujemo da se ove vrednosti i ne menjaju previše, sve imaju slične statistike.

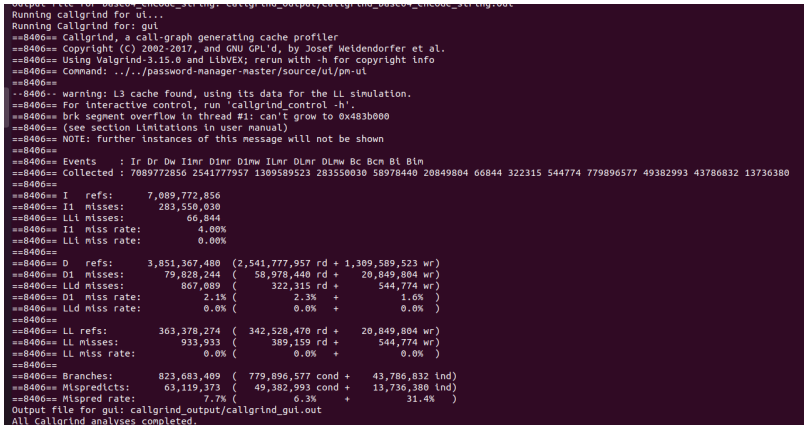


Slika 2: generate and copy

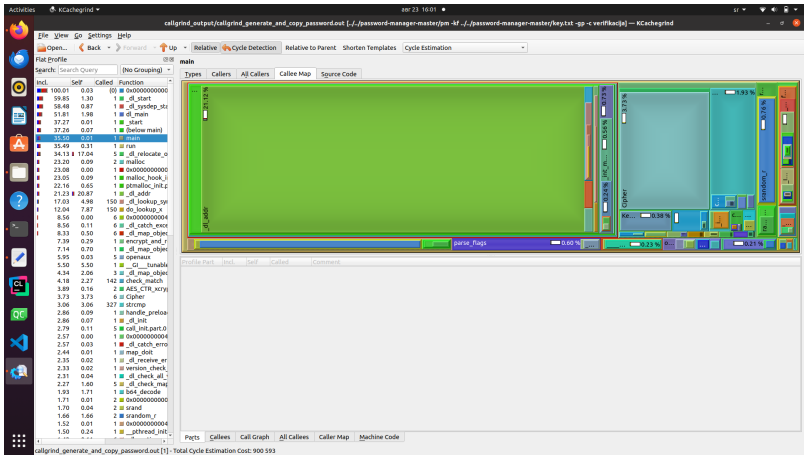
Pokretanjem gui aplikacije i korišćenjem iste dobijamo callgrind izveštaj, Slika 3. Ovde je pak I1 prmašaj keša viši 4.0%, dok je procenat promašaja keša ya podatke niži,kao i za pogrešnu predikciju grana (2.1% i 7.7 %).

Radi lakšeg praćenja funkcija, može se koristiti `kcachegrind file.out`. U kartici type je prikaz liste događaja i detalji o tome. Daje nam informacije o mapi poziva, grafu, listi poziva funkcija (Slika 4). Moguće je filtrirati izlaz ukoliko su nam od interesa source fajlovi, kao i kako se ponašaju funkcije u fajlu pm_xcrypt.c prilikom generisanja nove lozinke (Slika 5). Prikazana je mapa poziva, mogu'e je videti i graf poziva.

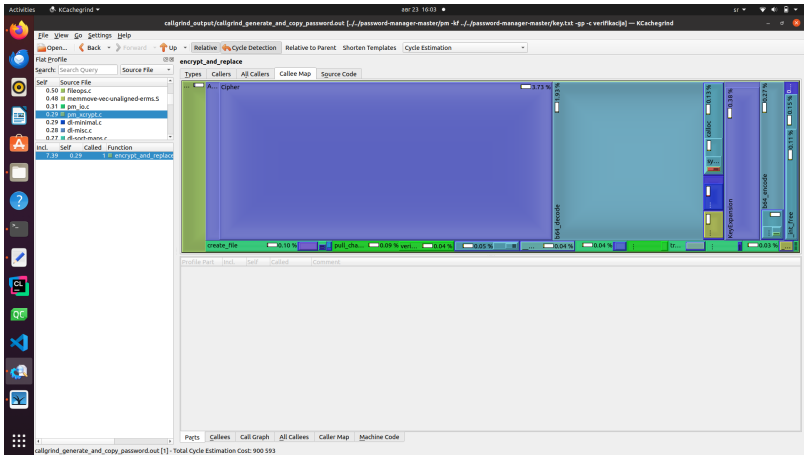
Korisno je primetiti da nema ciklusa i koliko vremena je iskoristila svaka funkcija.



Slika 3: generate and copy



Slika 4: Kcache: generate and copy



Slika 5: Kcache: generate and copy xcrypt

4 Cppcheck

Cppcheck je alat za statičku analizu koda napisanog u C/C++ jezicima. On pruža analizu koda kako bi otkrio greške i fokusira se na prepoznavanje neodređenog ponašanja i opasnih konstrukcija u kodu. Cilj je imati veoma mali broj lažno pozitivnih rezultata[1]. Cppcheck je dizajniran da može analizirati C/C++.

Za pokretanje Cppcheck-a iz konzole koristimo komandu `cppcheck file.c`. Mokuće je pokrenuti cppcheck za ceo folder `cppcheck path`, gde će cppcheck pokretati alat rekuezivno za ceo folder. Ukoliko ne želimo pokrenuti proveru za određeni fajl, moze se ručno navoditi putanja do svih izvornih fajlova koje želimo ispitati ili koristiti opciju `-i`, koja isključuje fajl na koji ima putanju navedenu u nastavku. Uobičajeno cppcheck koristi interni C/C++ parser, ali je moguće koristiti i clang parser. To se postiže `--clang` opcijom. Clang izlaz se svakako transformiše u Cppcheck format i izvršava standardna analiza.

Cppcheck je pokrenut posebno za konzolnu aplikaciju i za gui aplikaciju, skriptom `run_cppcheck.sh`. Za pregledniji izveštaj je moguće generisati xml fajl, a zatim html komadom `cppcheck -output-file=output.xml -xml password-manager-master/` (ovde za celu aplikaciju) i `cppcheck-htmlreport -file=output.xml -report-dir=report`.

[1]Lažno pozitivni rezultati su situacije u kojima alat za analizu ili testiranje prijavi grešku ili problem, iako stvarna greška ne postoji

Konzolna : izvestaj je čitljiv, za izvorni fajl dobija se linija i poruka. Na primer, mnoge promenljive u pm_aes.c imaju veći scope od potrebnog. U fajlu pm_b64.c postoje promenljive kojim su dodeljene vrednosti, a nisu korišćenje (a, b, c, d, j, l). Preporuka je da se uklone zbog čitljivosti koda(style). Cppcheck je pronašao i neka upozorenja (warnning),a to je da povratnu vrednost funkcije rand() nije iskorišćena. Shadowing je pojam koji označava da je promenljiva u užem scope-u nayvana isto kao i neka u širem. Na takave slučajeve nam je ukazao cppcheck.

Ui : Cppcheck je dao preporuke koje promenljive bi mogle biti deklarisane kao const, jer im se vrednost ne menja, to doprinosi čitljivosti koda. U c++ static clan funkcija je ona koja ne pristupa članovima klase, ako može da se izvrši bez pristupa i menjanja objekta, onda označavanjem kao static dobijamo na perfromansama i čitljivosti.

5 Clang

Clang je kompilator otvorenog koda za C familiju jezika. Koristi LLVM optimizator i generator koda. Clang statički analizator je deo Clang projekta.
Clang statički analizator koristi razne implementacije proveravača (engl. checkers) prilikom analize. Proveravači su kategorisani u familije - podrazumevani i eksperimentalni (alpha). Podrazumevani proveravači izvršavanju bezbednosne provere, prate korišenje API funkcija, traže mrtav kod i ostale logičke greške. Eksperimentalni (alpha) proveravači nisu podrazumevano uključeni pošto često daju lažne pozitivne rezultate.
`clang-tidy` je alat za C++ baziran na `Clang`-u, poznat kao "linter". Njegova svrha je da obezbedi proširiv okvir za dijagnostikovanje i ispravljanje tipičnih programskih grešaka, kao što su kršenja stilskih pravila, nepravilno korišćenje interfejsa ili greške koje se mogu zaključiti putem statičke analize. `clang-tidy` je modularan i pruža zgodan interfejs za pisanje novih provera (checks).
`clang-tidy` ima svoje provere i može takođe da pokreće provere iz `Clang Static Analyzer`-a. Svaka provera ima svoje ime i provere koje treba pokrenuti mogu se odabrati koristeći opciju `-checks=`, koja specificira listu sačinjenju od pozitivnih i negativnih (označenih sa `-`) globa, razdvojenih zarezima. Pozitivni globovi dodaju podskupove provera, dok negativni globovi uklanjaju te provere. Komanda `clang-tidy -list-checks` dakje spisak svih provera. Ovaj alat može automatski da reši neke stilske ili uobičajene greške prilikom programiranja.

Dijagnostika `clang-tidy` je namenjena da označi kod koji se ne pridržava standarda kodiranja ili je na neki način problematičan. Međutim, ako je kod poznat kao ispravan, može biti korisno da se upozorenje isključi. Neke `clang-tidy` provere pružaju specifičan način za isključivanje dijagnostike, na primer, `bugprone-use-after-move` može se isključiti ponovnim inicijalizovanjem promenljive nakon što je premeštena, `bugprone-string-integer-assignment` može se potisnuti eksplicitnim kastovanjem celog broja na `char`, `readability-implicit-bool-conversion` takođe može biti potisnut korišćenjem eksplicitnih kastova, itd.

Ovaj alat je pokrenut skriptom `run.sh`. Za funkciju koja koristi aes algoritam je pronađeno 6 upozorenja, ali sva su vezana za istu stvar. Naime, memcpy nije toliko siguran jer ne pruža sigurnosne provere uvedene C11 standardom, preporuka je zameniti je memcpy_s funkcijom. Ovoliki broj upoyorenja je zato sto je koristimo 6 puta, imamo informaciju tačno i u kojoj liniji. Fajlovi pm_b64.c, pm_glob.c i

pm_io.c nemaju nikakve upozorenja. pm_io.c i pm_parse.c takođe rade sa memcpy funkcijom. Fajl pm_parse.c ima najviše upozorenja i to:

Korišćenje `memcpy` bez provere granica Upozorenja se odnose na korišćenje funkcije `memcpy`, koja ne pruža sigurnosne provere granica. Funkcija `memcpy` može da izazove probleme ako se koristi za kopiranje podataka bez prethodne provere veličine, što može dovesti do prepisivanja memorije i sigurnosnih rupa. Preporučuje se da se `memcpy` zameni funkcijama koje pružaju bezbednosne provere, kao što je `memcpy_s` ako koristite C11 standard, ili sličnim funkcijama koje omogućavaju proveru dužine i granica.

```
Warning 1: /path/to/pm_parse.c:140:9
memcpy(aes_key, f.key.value, key_len < 32 ? key_len : 32);
Warning 2: /path/to/pm_parse.c:151:9
memcpy(aes_key, kf.start, kf.size < 32 ? kf.size : 32);
Warning 3: /path/to/pm_parse.c:225:13
memcpy(s.data, file.start, file.size);
```

Null pointer prosleđen kao argument Ova greška se javlja kada se funkciji `encrypt_and_replace` prosledi pokazivač koji može biti `NULL`. Ovo može dovesti do grešaka pri izvršavanju ako funkcija očekuje validan pokazivač. Potrebno je osigurati da su svi pokazivači koji se prosleđuju funkcijama inicijalizovani i da nisu `NULL` pre nego što ih koristite.

```
Warning: /path/to/pm_parse.c:240:44
encrypt_and_replace(&f, s, PM_STR(f.label.value), aes_key);
```

Polja `exists` se ne inicijalizuju pre povratka Postoje upozorenja da se neka polja, poput `f->help.exists` i `f->version.exists`, ne inicijalizuju pre nego što se koriste. Ovo može dovesti do nepredvidivog ponašanja ako se polja ne postave na odgovarajuće vrednosti pre nego što se upotrebe. Preporučuje se da se sva polja koja su deo struktura inicijalizuju pre nego što se koriste u logici programa.

```
Warning 1: /path/to/pm_parse.c:25:21
for (int i = 1; i < argc; i++)
```

```
Warning 2: /path/to/pm_parse.c:73:1
returning without writing to 'f->help.exists'
Warning 3: /path/to/pm_parse.c:87:16
if (f.help.exists)
```

Greške u logici petlji i uslova Ova upozorenja ukazuju na to da neka polja u strukturama (poput `f->data.exists`, `f->label.exists`, itd.) nisu postavljena na očekivane vrednosti pre nego što se koriste u uslovima. Potrebno je pregledati i osigurati da su svi uslovi u petljama pravilno postavljeni i da polja budu pravilno inicijalizovana pre nego što se koriste u logici.

```
Warning 1: /path/to/pm_parse.c:25:21
for (int i = 1; i < argc; i++)
Warning 2: /path/to/pm_parse.c:28:22
if (!f->data.exists && is_flag(argv[i], "-d", "--data"))
Warning 3: /path/to/pm_parse.c:30:28
else if (!f->label.exists && is_flag(argv[i], "-l", "--label"))
```

Još neka zapažanja slede.
Upozorenje u vezi sa fwrite funkcijom:
Na liniji 347 u fajlu pm_xcrypt.c, postoji uslov koji proverava da li je funkcija fwrite uspela da napiše sve podatke. Ako broj bajtova koje je fwrite napisala nije isti kao broj bajtova koji se očekivao (f.size), kod prelazi na deo na liniji 399 pomoću goto naredbe. Ovo može biti problematično jer ako fwrite ne uspe da napiše sve podatke, kod možda neće obraditi ovu grešku kako treba, što može dovesti do oštećenja podataka ili drugih problema.
Upotreba goto naredbe:
Na liniji 350, goto nardeba prebacuje kontrolu na liniju 399. Upotreba goto može otežati praćenje toka izvršavanja programa.

6 Literatura

[1] Milena Vujošević Janičić, Verifikacija softvera
[2]Materijali sa vežbi
[3]Valgrind
[4]gprof2dot
[5]cppcheck
[6]clang-tidy