ELEKTROTEHNIŠKI VESTNIK 88(1-2): 54–60, 2021
IZVIRNI STROKOVNI ČLANEK

Varen zagon nepreverjene programske kode v sistemu PIVO

Žiga Rojec

Univerza v Ljubljani, Fakulteta za elektrotehniko, Tržaška 25, 1000 Ljubljana, Slovenija
E-pošta: ziga.rojec@fe.uni-lj.si

Povzetek. PIVO (programerjevo interaktivno vadbeno okolje) je sistem za interaktivni študij algoritmičnega razmišljanja in programiranja, razvit na Fakulteti za elektrotehniko Univerze v Ljubljani. Uporabljamo ga za spodbujanje samostojnega študija pri predmetih, kjer se poučuje programiranje, primeren pa je tudi za izvajanje izpitov in tekmovanj. Študentje v sistemu prevzamejo nalogo, rešitev zanjo razvijejo v svojem okolju, na strežnik pa oddajo zaključeno izvorno kodo. Uporabnikova koda se na centralnem strežniku prevede, zažene in preizkusi. Oddana izvorna koda je pogosto nepopolna in potencialno škodljiva za neprekinjeno delovanje strežnika. V članku podrobno opisujemo načine za varen zagon nepreverjene kode, ki temeljijo na varnostnih mehanizmih jedra operacijskega sistema Linux. Z uporabo teh mehanizmov lahko strežnik varno in hitro souporablja več uporabnikov. Sistem PIVO so študentje dobro sprejeli, pozitivni učinki pri študiju pa so bili merljivi že po prvih semestrih uporabe.

Ključne besede: poučevanje na daljavo, programiranje, programska koda, vsebnik, Seccomp, škodljiva koda

A secure run of an untrusted program code in the PIVO system

PIVO (Programmer's Interactive Excercise Environment) is an interactive online practicing tool for algorithmical thinking and programming developed at the Faculty of Electrical Engineering, University of Ljubljana, Slovenia. It is used for independent study, excercising and examination in programming courses. Through the sistem, a student aquires a task, develops a solution in his/hers personal working environment and submits the source code to the server. The code is compiled, run and tested in real time. The submitted code can be incomplete and potentially harmful to the server. This paper proposes techniques for untrusted code sandboxing based on built-in Linux security mechanisms. Using the techiques, PIVO serves multiple users safely and fast. Among the students, the PIVO environment is well accepted, its impacts and positive study improvements are measurable already after only a few semesters of usage.

Keywords: distance learning, programming, program code, container, Seccomp, malicious code

1 UVOD

Računalniški sistemi so za sodobnega človeka skoraj neizogibni spremljevalci. Računalnik nam v različnih oblikah ne služi le kot način zabave, temveč je tudi močno, velikokrat nenadomestljivo delovno orodje. Da lahko to orodje v polnosti izkoristimo, ni dovolj le poznavanje prednaložene programske opreme. Za reševanje nekonvencionalnih problemov in koriščenje vseh računskih možnosti je potrebno znanje programiranja.

V tehniških, naravoslovnih in poslovodskih poklicih je programiranje postalo skorajda neizogibno. Prav zato je naloga višješolskega izobraževalnega procesa, da mlade kadre čim bolje pripravi na t. i. *algoritmični* način razmišljanja.

Poučevanje programiranja v visokem šolstvu je zahtevna naloga. Študentje, ki vstopajo v prvi letnik, prihajajo z najrazličnejšimi predznanji, od popolnega neznanja programiranja do takih, ki si z znanjem programiranja že služijo denar. Da študente v prvem letniku opremimo z minimalnim nivojem znanja algoritmičnega razmišljanja, na Fakulteti za elektrotehniko Univerze v Ljubljani pri poučevanju uporabljamo naslednje pristope:

1) predavanja konceptov,
2) vaje na primerih,
3) preverjanje samostojnega dela študentov (sprotne obveznosti),
4) preverjanje znanja (izpit).

Naj bo kakovost predavanj in vaj na še tako visoki ravni, je ključ do uspeha v znanju programiranja neizogibno povezana s tretjo točko – količino samostojnega dela študentov.

Programiranje je v osnovi ustvarjalen proces sinteze, ne samo analize. Zato je načinov do pravilnih rešitev zadanega problema veliko, lahko celo neskončno mnogo. V jeziku pedagogov: pravilnih odgovorov na eno izpitno vprašanje je lahko več, saj so možni različni načini pristopa k nalogi. Ker je vpis na tehniških fakultetah visok, imamo vsako leto opravka z okvirno štiristo študenti samo v prvem letniku [1]. V tem primeru postane preverjanje znanja zaradi časovnih in kadrovskih omejitev izjemno zahtevno. Za namen ocenjevanja je

Prejet 23. oktober, 2020
Odobren 8. januar, 2021

treba vsak program analizirati, po možnosti pognati v okolju, za katero je napisan, vstaviti vhodne in preveriti izhodne podatke ter dodeliti ocene uspešnosti. To delo je brez napak in v doglednem času težko opraviti ročno.

Ko začnemo razvijati sistem, ki bo na osebnem računalniku avtomatično zaganjal in preverjal oddano kodo, se je treba zavedati, da je študentska koda lahko nepopolna. Naštejmo nekaj najpogostejših primerov, ko najrazličnejša koda, ki jo zaporedoma preizkušamo, povzroči težave:

- vsebuje sintaktične napake (ni prevedljiva),
- vsebuje neskončne zanke,
- naslavlja globalne spremenljivke,
- znakovno kodiranje izvorne kode se ne ujema s sistemskim kodiranjem.

Študentje lahko takšno kodo oddajo v dobri veri, da se bo med avtomatskim pregledovanjem izkazala za ustrezno, še zlasti če se med tem procesom izvedejo dodatni testi, ki študentu prej niso bili prikazani. Obstaja pa tudi drug spekter težav, povezanih z zaganjanjem tuje kode – lahko je namensko škodljiva. Za namen upočasnitve, onemogočanja ali vohunjenja na sistemu lahko:

- prekomerno izrablja sistemske vire,
- ustvarja ali bere sistemske datoteke,
- ustvarja nove procese,
- komunicira z zunanjim omrežjem,

in podobno. Če sistem, ki bo pregledal kodo, poženemo le nekajkrat na semester, je stabilnost sistema dokaj enostavno zagotoviti z ustrezno skripto, ki omejuje npr. čas izvajanja določenega procesa. Vse preostalo lahko rešimo z zaganjanjem sistema v virtualnem računalniku in ustvarjanjem varnostnih kopij slike tega virtualnega računalnika. Za zmanjšanje težav v takih sistemih je proti avtorjem namensko škodljive kode treba uveljavljati posebne določbe izpitnega reda, npr. negativne točke ali dodatno delo.

Če razvijamo sistem, ki bo deloval *interaktivno* in pregledoval ter ocenjeval naloge *v realnem času* (na primer na izpitu), pa je arhitekturo takega sistema treba skrbno načrtovati. Sistem mora ostati nedotaknjen in procesi morajo delovati neprekinjeno – tako v primeru zagona dobre kot škodljive kode.

V pričujočem prispevku bomo predstavili naš pristop za izgradnjo sistema, v katerem je mogoče varno zagnati programsko kodo, ki ji *a priori* ne zaupamo, preizkusiti njeno delovanje in ohraniti nedotaknjenost sistema.

2 Obstoječi sistemi avtomatskega preverjanja programskih rešitev

V tem kratkem podpoglavju navedimo nekaj obstoječih sistemov, ki so namenjeni zagonu nepreverjene kode in imajo predvsem pedagoški namen.

V grobem lahko sisteme razdelimo na dva tipa, in sicer glede na to, kje se oddana koda izvede; to se lahko zgodi:

- v uporabnikovem brskalniku ali
- na gostiteljskem strežniku.

Na svetovnem spletu lahko najdemo nekaj strani, kjer je mogoče programsko kodo zagnati prek brskalnika. Spletna stran w3schools.com, na primer, uporabnikom strani omogoča, da preizkušajo kratke programe v jeziku JavaScript – ti se izvedejo v uporabnikovem brskalniku. Stran, ki izvaja kodo JavaScript v brskalniku, je dokaj enostavna za implementacijo, saj večina spletnih brskalnikov že v osnovi podpira interpretacijo tega jezika, zato jih na spletu najdemo veliko, npr. [6], [7], [8] in številne druge.

Na spletu prav tako najdemo sisteme, ki ponujajo zagon uporabnikove kode tudi v drugih programskih jezikih. Pri tem velja omeniti, da so v preteklosti nekateri sistemi delovanje programske kode le simulirali z jezikom JavaScript v brskalniku, kar znatno omejuje možnosti in smiselnost preverjanja delovanja kode. Če želimo program zares preizkusiti, je treba kodo zagnati na strežniku, v okolju, ki že v osnovi interpretira oz. prevede in zažene zapisano kodo. Primer takega sistema najdemo na spletni strani codechef.com, ki podpira izvajanje kode več kot 50 različnih programskih jezikov [9]. Indijska spletna stran je ustvarjena tako, da lahko gosti tudi programerska tekmovanja. Codecademy.com je stran, ki nudi interaktivne seminarje za učenje različnih programskih jezikov, tudi v tem sistemu lahko izvedemo kodo na oddaljenem strežniku [10]. Omeniti velja še codepad.org, ki je namenjen predvsem shranjevanju in medsebojnemu deljenju kratke programske kode [11]. Zanimivost sistema je, da za nalaganje in izvajanje kode ne zahteva registracije uporabnika, saj naj bi njihovi varnostni ukrepi v veliki meri onemogočali zlorabe sistema in omogočali njegovo neprekinjeno delovanje. Enega od redkih odprtokodnih sistemov za interaktivno učenje programskega jezika C++ najdemo na omrežju IRC. Gre za Geordi, t. i. *bot*-a, ki prevaja in poganja koščke kode, ki jih v klepetalnik IRC vnaša uporabnik. Za zagotavljanje varnosti sistem Geordi uporablja poseben Linuxov mehanizem Seccomp [14].

V slovenskem prostoru je znan portal putka.si (Preverjanje učinkovitosti, temeljitosti in korektnosti algoritmov) Zavoda za računalniško izobraževanje Ljubljana, namenjen poučevanju programiranja in izvedbi tekmovanj. V Putki so naloge dostopne prek spletne strani, rešitev pa uporabnik/študent naloži v obliki datoteke izvorne kode. Koda se na strežniku prevede, ob izvajanju pa so onemogočeni vsi kritični sistemski klici, s čimer upravljavci zagotavljajo varnost strežnika. Onemogočeno je tudi npr. ustvarjanje datotek in delo z njimi, upravljavci sistema pa lahko za posamezne naloge omogočijo določene kritične sistemske klice. Na ta način je sistem sicer fleksibilen, vendar dojemljiv za varnostne probleme.

Med obstoječimi interaktivnimi učnimi sistemi za programiranje v domačem prostoru izstopa Projekt Tomo

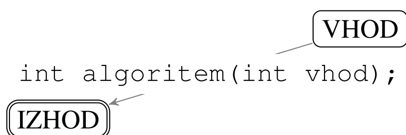
[2]. Za podporo poučevanju programiranja ga uporabljajo na več srednjih šolah in fakultetah. Tomo deluje tako: Uporabnik se registrira na spletno mesto, kjer lahko izbira med nalogami, ki so mu na voljo. Strežnik uporabniku servira nalogo, zapisano v obliki skriptne datoteke (npr. tipa .py za Python). V skripto uporabnik vnaša programsko rešitev in jo požene na lokalnem računalniku. Del skripte, ki jo servira Projekt Tomo, vsebuje tudi vse ukaze, ki izhode uporabnikovega programa pošlje nazaj na strežnik za ocenjevanje. Poudariti je treba, da se koda izvede zgolj lokalno, strežnik pa skrbi le za preverjanje pravilnosti izhodov programa. Iz skripte lahko torej že nespreten uporabnik vnaprej izve za vse teste, ki bodo izvedeni na njegovi kodi. V nekaterih primerih je tak način dela zelo dobrodošel (npr. za podporo pedagoškemu delu pri laboratorijskih vajah), velikokrat pa se uporabniki zato osredotočijo zgolj na zadovoljevanje testnih primerov (ker so vsi znani), in ne na razvoj programa, ki bi celostno reševal zadani problem [3], [4]. Prav zato je sistem manj primeren za končno preverjanje znanja in izvedbo tekmovanj.

3 Sistem PIVO

Zaradi nekaterih pomanjkljivosti obstoječih sistemov, predvsem pa zaradi enostavnejšega vzdrževanja *v hiši* smo se pri avtomatskem pregledovanju nalog iz programiranja odločili za vzpostavitev lastnega sistema, ki najbolje ustreza načinu dela pri poučevanju programiranja na naši fakulteti. Ker uporabniki – programerji-začetniki – odziv sistema prejmejo v zelo kratkem času, smo ga poimenovali *Programerjevo interaktivno vadbeno okolje* (*PIVO*).

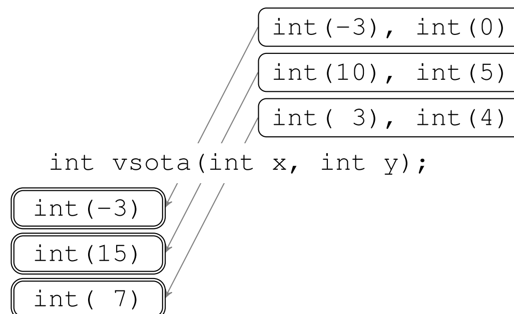
3.1 Formulacija problema

Preverjanje nalog iz programiranja je mogoče avtomatizirati, če probleme (naloge) ustrezno formuliramo. Pri programiranju algoritmov pogosto želimo določene vhodne podatke obdelati, kot rezultat pa vrniti bodisi izluščen kos informacije bodisi spremenjene vhodne podatke. Primerna formulacija naloge je na primer, da študentje napišejo definicijo ustrezne programske funkcije (oz. metode, podprograma) v dogovorjenem programskem jeziku (slika 1).



Slika 1: Računalniški program: vhod-algoritem-izhod.

Taka formulacija naloge se ujema tudi s splošnimi smernicami pisanja dobre kode [12]. Ker so posamezni sklopi kode zapisani modularno, je program namreč veliko bolje berljiv ter primeren za razširjanje in vzdrževanje. Od študentov oz. uporabnikov sistema PIVO tako vedno zahtevamo, da želeni algoritem zapišejo v obliki funkcije. Če je naloga zastavljena tako, da sta tip in število podatkov, ki jih program sprejme in vrne, točno določena, lahko definiramo testne nize podatkov, ki jih na oddanem študentskem programu preizkusimo (slika 2).



Slika 2: Preizkušanje enostavnega računalniškega programa na testnih podatkih – primer vsote dveh števil.

Prav tako uporabnikom že v navodilih za izvedbo naloge predstavimo celoten program, ki se bo zagnal ob testiranju. Tak pristop ima vsaj dve prednosti: uporabniki imajo vse informacije o tem, kako razvijati program na svojem sistemu; uporabniki vedo, na kakšen način bo njihov program preizkušen. Vnaprej navedemo vse dovoljene knjižnice, globalne spremenljivke in način izpisa rezultatov na standardni izhod. Spodaj je prikazan primer podane kode za nalogo preproste vsote dveh celih števil v programskem jeziku C.

```
#include <stdio.h>
int vsota(int, int); // Prototip funkcije.
int main(){
    int x, y;

    // Tukaj se bodo zaporedoma
    // izvedli testni klici.
    return 0;
}

// Od tu naprej bo
// v prevajanje kode
// vkljuceno vase delo.
```

3.2 Preizkušanje programa

Testni klici zahtevane funkcije skupaj s pričakovanimi izpisi na standardni izhod so na spletni strani sistema PIVO podani ločeno:

```
x = 3, y = 4;
printf("%d", vsota(x, y)); // 7

x = 10, y = 5;
printf("%d", vsota(x, y)); // 15
```

Razlog za ločeno podajanje testnih klicev je pedagoški; izkušnje so pokazale, da študentje lažje ugotovijo pričakovano delovanje funkcije, če so testni nizi grafično blizu navodil za delovanje pričakovanega dela kode. Ob preizkušanju kode, ki je oddana na

strežnik, se k zgornjem nizu testov dodajo še taki, ki študentu/uporabniku vnaprej niso bili navedeni pri prevzemu naloge, pa vendar spadajo v definicijsko območje naloge (npr. vsota mora delovati tudi za negativna števila).

```
x = -3, y = 0;
printf("%d", vsota(x, y)); // -3
```

S tem ko vsi testi niso vidni, se študentje/uporabniki resnično osredotočijo na reševanje problema, in ne samo na zadovoljevanje testnih primerov.

3.3 Oddaja kode na strežnik

Študent/uporabnik razvito izvorno kodo odda v temu namenjeno tekstovno okno na spletni strani. V nasprotju z nalaganjem tekstovne datoteke ima *lepljenje* kode v spletni obrazec nekaj prednosti: za študenta je hitrejše ter odpadejo problemi z izvornim kodiranjem tekstovne datoteke, ki so se izkazali za veliko težavo v prvih različicah sistema. Primer oddane uporabnikove kode, ki vrne pravilne rezultate:

```
int vsota(int stevilo1, int stevilo2){
 int r; stevilo1 += stevilo2;
 r = stevilo1;
 return r;
}
```

Večina sistema PIVO je skupaj s pripadajočo spletno stranjo izdelana v programskem jeziku Python, s pomočjo knjižnice Django, popularnega sistema za razvoj in vzdrževanje strežniškega zaledja [16]. V času nastanka tega prispevka PIVO podpira preizkušanje kode v programskih jezikih C in JavaScript (Node.js).

4 Varen zagon tuje kode

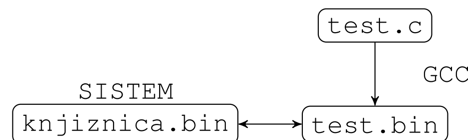
Največji izziv pri zagonu tuje kode je zagotavljanje varnosti in s tem neprekinjenega delovanja strežnika, čemur bomo posvetili nekaj več besed. Pri varnosti v našem sistemu smo ubrali dokaj rigorozen pristop. Predvidevali smo, da je vsaka koda lahko namensko škodljiva in mora kot taka imeti čim bolj omejen dostop do preostale računalniške infrastrukture. V tem poglavju bomo poskušali opisati način varnega zagona kode, napisane v programskem jeziku C.

4.1 Inicializacija knjižnic

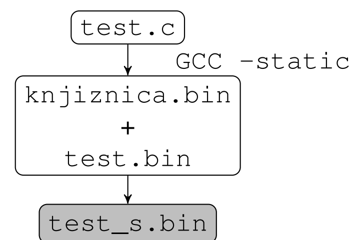
Če razmišljanje začnemo od spodaj navzgor, lahko z vidika varnosti sprva zagotovimo, da ima vsak program na voljo *kvečjemu* tiste vire (programske ali strojne), ki jih nujno potrebuje za svoje delovanje. To zahtevo lahko izpolnimo, če poskrbimo, da se ob prevajanju programa upoštevajo le knjižnice/moduli, ki smo jih vnaprej navedli v testnem programu – zagotoviti je treba, da uporabnik sistema ne more vključiti dodatnih knjižnic. To je mogoče narediti že v fazi pred prevajanjem programa z enostavno analizo izvorne kode in odstranitvijo ključnih ukazov za inicializacijo knjižnic.

4.2 Prevajanje

Sledi prevajanje izvorne kode. Za kodo v programskem jeziku C uporabimo prevajalnik GCC, ki mu nastavimo opcijo za statično prevajanje. To pomeni, da bo koda prevedena v strojni jezik skupaj z zahtevanimi knjižnicami; zagon programa je tako neodvisen od nameščenih sistemskih knjižnic.



Slika 3: Prevajanje in souporaba sistemskih knjižnic.



Slika 4: Statično prevajanje – knjižnica je vključena v binarni del programa.

Če se zgodi, da je izvorna koda neprevedljiva, uporabniku vrnemo sporočila prevajalnika. Ker prevajalnik GCC velikokrat sporoča dele izvorne kode, v katerih bi lahko bila napaka, pred tem iz sporočil odstranimo morebitne (skrite) testne klice in morebitno vidno direktorijsko strukturo strežnika.

4.3 Zaklepanje procesa

Preden preveden program zaženemo na strežniku, moramo poskrbeti, da ta ne bo zahteval več sistemskih virov, kot je potrebno [13]. Ne pozabimo, da je prevedeni progam še vedno lahko nepopoln ali pa namensko škodljiv. Tipičen primer nepopolnega programa je, da ta vsebuje ponavljalne stavke, ki se nikoli ne izvedejo do konca. Tako program zaseda procesorski čas in drugi procesi na strežniku zastanejo, kar je treba preprečiti. Prav tako je treba omejiti zahteve po prevelikem kosu pomnilnika (angl. *memory allocation*), spreminjanju direktorijske strukture, ustvarjanju preveč novih procesov in podobno.

Vse te zahteve lahko združimo v program, ki procesu *zaklene* (angl. *lockdown*) nedovoljene sistemske klice in omeji vire. V operacijskem sistemu Linux lahko to storimo z uporabo funkcij iz knjižnice `seccomp` (Secure Computing Mode) [14]. Seccomp je mehanizem v jedru Linuxa, ki omogoči prehod procesa v t. i. varno stanje, kjer so v osnovi onemogočeni vsi sistemski klici razen nekaj nujnih (sporočilo o stanju in izhodu procesa, branje in pisanje na standardni vhod/izhod). Toda če upoštevamo vse blokade, ki jih `seccomp` lahko izvede

na procesu, se lahko kaj hitro zgodi, da bo program, ki ga želimo preizkusiti, neuporaben. Če je npr. ustvarjen za obdelavo tekstovnih datotek, mu je treba omogočiti branje in/ali pisanje datotek. Naslednji set pravil za `seccomp` omogoči procesu odpiranje (`open`) in zapiranje (`close`) datotek, striktno pa onemogoči kloniranje procesov (`vfork` ali ustvarjanje novih map (`mkdir`).

```
#include <seccomp.h>
scmp_filter_ctx const ctx =
                    seccomp_init(SCMP_ACT_TRAP);
seccomp_rule_add(ctx,SCMP_SYS(open),
                              SCMP_ACT_ALLOW,0);
seccomp_rule_add(ctx,SCMP_SYS(close),
                              SCMP_ACT_ALLOW,0);
seccomp_rule_add(ctx,SCMP_SYS(vfork),
                              SCMP_ACT_ERRNO(0),0);
seccomp_rule_add(ctx,SCMP_SYS(mkdir),
                              SCMP_ACT_ERRNO(0),0);
```

Pri omejevanju porabe sistemskih virov nam pomaga `setrlimit(2)` sistemski klic v Linuxu. Naslednji primer prikazuje omejevanje uporabe procesorskega časa `RLIMIT_CPU` na 2 sekundi (mehki pogoj) oziroma 3 sekunde (strogi pogoj).

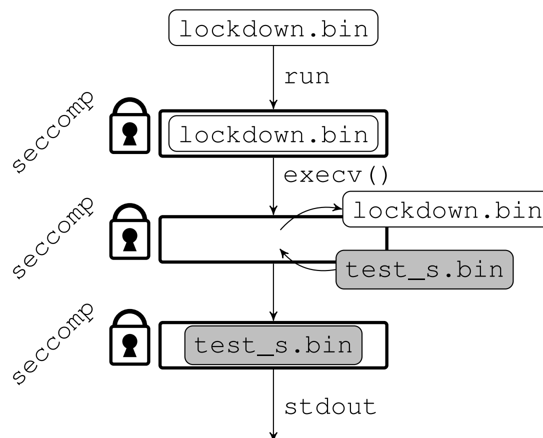
```
#include <sys/time.h>
#include <sys/resource.h>
setrlimit(RLIMIT_CPU, 2, 3);
```

Na obstoječi proces se torej nastavi časovna omejitev – če je proces končan v manj kot 2 sekundah, se blokada procesa ne sproži, če preseže mejo mehkega pogoja, se mu pošlje signal `SIGXCPU` (presežen procesorski čas) in, nazadnje, če proces po preseženi meji strogega pogoja še vedno obstaja, ta prejme signal `SIGKILL`. Na podoben način lahko omejimo tudi največjo velikost datotek, ki jih proces lahko ustvari, in velikost pomnilnika (v bajtih), ki ga lahko rezerviramo na RAM-u.

```
int slim = 10*1024*1024;
int hlim = 12*1024*1024;
setrlimit(RLIMIT_FSIZE, slim, hlim);
setrlimit(RLIMIT_DATA, slim, hlim)
```

Ko so vse opisane omejitve za proces nastavljene, lahko zaženemo novi, zaupanja nevreden proces – kodo uporabnika sistema PIVO. To storimo tako, da v programu, za katerega smo nastavili omejitve, pokličemo ukaz `execv`, ki zažene binarno datoteko, katere pot podamo v parametrih. Ukaz `execv` ne ustvari novega procesa, temveč pod istim `PID` in enakimi pogoji (omejitvami sistemskih virov) zažene drug binarni program (slika 5). Spodnji klic funkcije `execv` zažene proces, ki smo ga podali kot prvi argument pri klicu programa `lockdown` (`argv[1]`), in kazalec usmeri na preostanek podanih parametrov (kazalec `argv + 1`), ki služijo kot vhodni podatek v novi proces.

```
execv(argv[1], argv + 1);
```



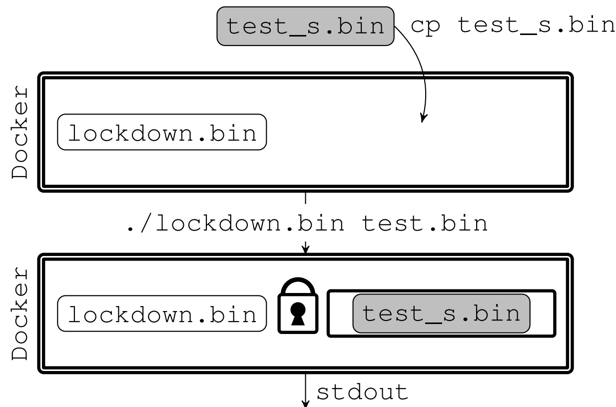
Slika 5: Zaklepanje procesa – program `lockdown.c` zaklene dovoljenja za proces, ukaz `execv()` nadomesti program, ki trenutno teče, z uporabnikovo izvršilno kodo `test_s.bin` pod isto (starševsko) procesno identiteto. Ob preverjanju delovanja `test_s.bin` spremljamo vsebino standardnega izhoda.

4.4 Zagon programa v vsebniku

Zavoljo dodatne varnosti celoten proces zaklepanja in zagona procesa zaženemo v vsebniku Docker. Docker je orodje za razvoj aplikacij, v katerega vsebnik namestimo vse, kar bi naš program lahko potreboval (izvršilne datoteke, sistemske knjižnice, direktorije itd.) [15]. Gre za "lahkokategorni" način virtualizacije, kjer je program zagnan znotraj vsebnika, med delovanjem pa je od gostiteljevega operacijskega sistema popolnoma izoliran – delita si le jedro operacijskega sistema.

Docker ima že vgrajene nekatere varnostne mehanizme, ki onemogočajo "pobeg" procesa iz nadzorovanega okolja. Glavni namen uporabe vsebnika v našem primeru je simulacija praznega operacijskega sistema – na ta način zagnani proces nima na voljo sistemskih direktorijev ali knjižnic. Za zagon uporabnikove kode imamo pripravljen vsebnik, ki vsebuje le izvršilno datoteko `lockdown.bin`. Ko uporabnik pošlje zahtevek za zagon kode, se statično prevedena izvršilna datoteka (izvršilna datoteka že vključuje potrebne knjižnice, glej poglavje 4.2) kopira v pripravljeni vsebnik. Tako je morebitno škodljivi proces dodatno izoliran od sistema, za zagon pa ima na voljo izključno datoteke, ki so na voljo v tem vsebniku (`lockdown.bin` in `test.bin`). Poudariti moramo, da ni treba, da je v vsebniku nameščen kakršenkoli drug program, niti Linuxova lupina Shell ali Bash, prek katerih bi lahko sistemu, nameščenemu v Docker, pošiljali sistemske klice.

Proces zaženemo z `lockdown.bin`, ki zaklene dostop do sistemskih strojnih virov, kot je opisano v prejšnjem podpoglavju (4.3). Ko se proces zažene, lahko spremljamo njegov zapis na standardni izhod. Slika 4.4 prikazuje proces zagona izvršilne datoteke v izoliranem okolju.



Ko se prek protokola HTTP zgodi zahteva po prevajanju in zagonu kode, se na strežniku ustvari kopija praznega vsebnika. Ustvarjanje kopij vsebnika povzroča znaten časovni zamik (nekaj 100 ms), vendar pa omogoča, da se vsak program zažene v ločenem okolju, v simulaciji popolnoma praznega operacijskega sistema.

Alternativa temu pristopu bi bila uporaba enega samega vsebnika Docker, v katerega po metodi cevovoda (*pipe-line*) vnašamo izvršilne datoteke in jih preizkušamo zaporedoma. Ta pristop na strežniku deluje hitreje, vendar povzroča druge težave; pojavi se na primer vprašanje, kako posameznim procesom zagotoviti enakovredno okolje (prazen sistem), če npr. procesi ustvarjajo ali berejo datoteke, se sklicujejo na različne sistemske knjižnice in podobno.

4.5 *Navidezni računalnik*

Za namen doseganja visoke varnosti in enostavnega upravljanja je sistem skupaj s strežnikom nameščen v virtualni računalnik. Tako se enostavno vzpostavlja varnostne posnetke sistema (*snapshots*), morebitna škodljiva koda pa ima okrog vsebnika še dodatno varnostno bariero, ki preprečuje, da bi imela dostop do centralnega računalniškega sistema.

4.6 *Preizkus delovanja*

4.6.1 Robustnost sistema: Varnost zaledja sistema PIVO smo preizkusili z naborom znanih odprtokodnih virusov in škodljivih programov. Omeniti velja, da je večina dostopne škodljive programske opreme sicer prilagojena delovanju v operacijskem sistemu Windows, tako da je za namene testiranja (oziroma škodovanja sistemu) to opremo treba prilagoditi, da bi lahko škodovala sistemu Linux. Preizkusili smo oddaljeno ugašanje računalnika, rekurzivne zahteve po velikih kosih pomnilnika, rekurzivno kopiranje obstoječih procesov in podobno – v vseh primerih je sistem delovanje škodljive kode ustavil, preden bi prišlo do kakršnegakoli opaznega vpliva na operacijski sistem virtualnega strežnika. Spomnimo naj, da v osnovni različici sistema PIVO v vsebniku Docker, ki je dodeljen vsakemu uporabniku, ni nameščen niti Linuxov Shell, s pomočjo katerega bi lahko prek sistemskih klicev nenadzorovano (mimo pravil sistema Seccomp) upravljali z jedrom operacijskega sistema.

4.6.2 Uporaba v praksi: Do današnjega dne smo na Fakulteti za Elektrotehniko Univerze v Ljubljani sistem PIVO uporabljali za namen izvajanja obveznih domačih nalog pri predmetih iz programiranja na univerzitetnem in višješolskem strokovnem programu. Večjih poskusov zlorab sistema do zdaj nismo opazili – vsak študent je v okolje PIVO prijavljen z univerzitetno identiteto, pri vsaki nalogi pa je na voljo le omejeno število poskusov. Oboje zmanjšuje privlačnost nalaganja namensko škodljive kode. Kljub vsemu je tako vzpostavljen sistem pripravljen na zahtevnejšo rabo, na primer za neregistrirane uporabnike ter izpite in tekmovanja.

5 ZAKLJUČEK

Ročno ocenjevanje znanja programiranja je dolgotrajen proces. Med izobraževalnim procesom ga je mogoče avtomatizirati na točki, ko študentje usvajajo principe t. i. algoritmičnega razmišljanja. Takrat je mogoče preverjanje znanja omejiti na vstavljanje podatkov v algoritem in analizo izhodnih podatkov. Študentje dosežejo najboljše rezultate, če imajo v času študija na voljo hitre povratne informacije o svojem trenutnem uspehu. Zato smo pri predmetih iz programiranja na Fakulteti za elektrotehniko Univerze v Ljubljani razvili sistem za interaktivno učenje programiranja PIVO (*Programerjevo interaktivno vadbeno okolje*), ki uporabniku v trenutku poda povratno informacijo o pravilnosti delovanja oddanega algoritma. Uporabnikova koda se izvede na centralnem strežniku, zato je varnosti izvajanja posvečena posebna pozornost. V članku smo opisali tehnično rešitev za hiter in varen zagon nepreverjene kode, ki temelji na tehnologiji Seccomp ter vsebniku Docker. Sistem je razvit tako, da se uporabnikova koda v strojni jezik prevede z največ tistimi viri, ki so nujno potrebni za delovanje programa. Posebni varnostni mehanizmi omejijo delovanje procesa na točno določene funkcionalnosti, ki jih določena naloga od uporabnika zahteva. Sistem v prvih letnikih študija na Fakulteti za elektrotehniko v praksi uporabljamo več let. Rezultati so spodbudni. Več kot 1000 študentov je dokazalo, da se je splošno znanje osnov programiranja tako izboljšalo, da bi bilo treba v naslednjih letih za vzdrževanje Gaussove krivulje uspeha na izpitu prag zahtevanega znanja celo dvigniti. Znova se je pokazalo, da je znanje programiranja odvisno predvsem od praktičnega dela in vloženega truda vsakega študenta.

Sistem PIVO se je kot dober pripomoček izkazal še posebej v času izvajanja študija na daljavo. Študentom je omogočil interaktiven študij na širokem spektru programskih problemov brez omejitev (časovnih, kadrovskih, prostorskih), ki bi sicer obstajale v fizičnem laboratoriju. Z uporabo sistema PIVO je bilo možno

nadomestiti vsaj omejen del laboratorijskih vaj, ki bi jih bilo sicer potrebno opraviti na fakulteti.

LITERATURA

[1] Fakulteta za elektrotehniko Univerze v Ljubljani, *Letno poročilo 2018*, Fakulteta za elektrotehniko, 2019. URL: http://www.fe.uni-lj.si/mma/letno-2018/2019030613142714/

[2] G. Jerse, M. Lokar, *Learning and Teaching Numerical Methods with a System for Automatic Assessment*, INTERNATIONAL JOURNAL FOR TECHNOLOGY IN MATHEMATICS EDUCATION, 2017.

[3] M. Pretnar, *Spletna storitev za poucevanje programiranja*, VIVID 2014 C P KRANJ, strani 194-201, Fakulteta za organizacijske vede, Kranj, 2014.

[4] M. Lokar, M. Pretnar, *A low overhead automated service for teaching programming*, 15 KOL CALL C COMP E, strani 132-136, 2015. URL: https://doi.org/10.1145/2828959.2828964

[5] W3schools.com, dostopno 9. jan. 2020. URL: https://www.w3schools.com/

[6] fireship.io, dostopno 9. jan. 2020. URL: https://fireship.io

[7] js.do, dostopno 9. jan. 2020. URL: https://js.do

[8] playcode.io, dostopno na: https://playcode.io, jan. 2020.

[9] Directi, *codechef.com*, dostopno na: https://www.codechef.com/ide, jan. 2020.

[10] *codecademy.com*, dostopno 10. jan. 2020. URL: https://www.codecademy.com

[11] S. Hazel, *codepad.org*, dostopno 10. jan. 2020. URL: http://codepad.org

[12] I. Fajfar, *Start programming using HTML, CSS, and JavaScript*, Boca Raton ; London ; New York : CRC Press, Taylor & Francis Group, cop. 2016.

[13] Eelis, *Geordi: IRC C++ eval bot*, Javna domena, dostopno na https://github.com/Eelis/geordi, Jan. 2020.

[14] A. Arcangeli, *Seccomp in Linux (4.19)*, dostopno na: https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html, jan. 2020.

[15] Docker, Inc., *Docker*, dostopno na https://docker.com, feb. 2020.

[16] Django (Version 1.5), *Django*, dostopno na https://djangoproject.com, feb. 2020.

Žiga Rojec je leta 2014 magistriral, leta 2018 pa doktoriral na Fakulteti za elektrotehniko Univerze v Ljubljani. Od leta 2014 je zaposlen kot asistent na Fakulteti za elektrotehniko. Raziskovalno se ukvarja z inovativnimi postopki za avtomatsko sintezo topologij električnih vezij z uporabo evolucijskih algoritmov. S študenti izvaja vaje s področja programiranja in analize električnih vezij. Vzdržuje gručo strežnikov Linux, ob tem pa tudi razvija in nadgrajuje sistem PIVO za interaktivno spletno učenje programiranja.