CARNet
HRVATSKA AKADEMSKA I ISTRAŽIVAČKA MREŽA
CROATIAN ACADEMIC AND RESEARCH NETWORK

Besplatni alati za otkrivanje pogrešaka u programskom kodu

CCERT-PUBDOC-2009-08-274

+CERT.hr u suradnji s LSS

Sigurnosni problemi u računalnim programima i operativnim sustavima područje je na kojem CARNet CERT kontinuirano radi.

Rezultat toga rada je i ovaj dokument, koji je nastao suradnjom CARNet CERT–a i LS&S–a, a za koji se nadamo se da će Vam koristiti u poboljšanju sigurnosti Vašeg sustava.

CARNet CERT, www.cert.hr

Nacionalno središte za **sigurnost računalnih mreža** i sustava.

LS&S, www.LSS.hr

Laboratorij za sustave i signale pri Zavodu za elektroničke sustave i obradbu informacija Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Sadržaj

1. Uvod

Velik broj svakodnevno otkrivenih sigurnosnih ranjivosti ukazuje na činjenicu da se na sigurnost još uvijek ne obraća dovoljno velika pozornost prilikom razvoja programa. Pisanje sigurnih programa nije jednostavan zadatak – broj sigurnosnih propusta koje je moguće napraviti je prilično velik. Usprkos tome, pokazalo se kako je programerska zajednica sklona ponavljanju istih pogrešaka. Primjerice, preljev međuspremnika (engl. *buffer overflow*) kao sigurnosni propust aktualan je već nekoliko desetljeća, no ipak je još uvijek jedan od najčešće pojavljivanih propusta (CWE TOP 25[6]). Većina novootkrivenih propusta spada u neku od poznatih i već vrlo dobro istraženih kategorija ranjivosti. Ta je činjenica potaknula razvoj automatiziranih alata za pronalazak najčešćih pogrešaka u programskom kodu.

U ovom dokumentu prikazana su tri besplatna alata za *statičku analizu programa*: flawfinder, RATS (Rough Auditing Tool for Security) i splint. Statička analiza je oblik analize programa koja se provodi bez samog izvršavanja programa (takva se analiza naziva *dinamičkom*), najčešće nad nekom vrstom izvornog ili objektnog koda. Svi navedeni alati, izuzev splinta (koji provodi nešto dublju analizu), vrše samo *leksičku* analizu programskog koda, što znači da ne prate tijek izvođenja naredbi, već samo traže predefinirane uzorke u kodu. Alati sadrže bazu potencijalno nesigurnih funkcija s pridjeljenim razinama rizika te upozoravaju na njihovu upotrebu u programskom kodu.

Većina primjera u dokumentu odnosi se na programski jezik C. Kao jezik niske razine, C je inherentno nesiguran jezik. Čak i sama standardna C biblioteka sadrži neke nesigurne funkcije (primjerice `strcpy()`, `gets()` i sl.) čija se upotreba ne preporuča.

2. Najčešće ranjivosti u programima

CVE (Common Vulnerabilities and Exposures - http://cve.mitre.org) projekt održava listu od 25 najopasnijih programskih grešaka (CWE Top 25). Prema toj listi, pogreške su ugrubo podijeljene u sljedeće tri skupine:

- **nesigurna interakcija između komponenti** – ranjivosti iz ove skupine vezane su uz nesigurne načine na koje se podaci razmjenjuju između pojedinih komponenti, modula, programa ili sistema (npr. neispravna validacija ulaznih podataka, prijenos osjetljivih podataka u čistom tekstu, *race condition* propusti i sl.) ,
- **riskantno upravljanje resursima** – skupina uključuje ranjivosti vezane uz neispravne načine stvaranja, korištenja, prijenosa ili oslobađanja sistemskih resursa, kao što su curenje memorije, dereferenciranje *null* pokazivača, preljevi međuspremnika itd.
- **nepotpune metode zaštite** – ranjivosti iz ove skupine vezane su uz neispravno korištene, zloupotrebljene ili zanemarene načine zaštite (npr. izvršavanje s nepotrebno visokim privilegijama, neispravna kontrola pristupa, izvršavanje provjera na klijentskoj strani koje bi se trebale obavljati na serverskoj itd.)

U nastavku je detaljnije prikazano nekoliko najčešćih ranjivosti.

2.1. Neispravna validacija ulaznih podataka

Svaki imalo složeniji računalni program prima određene podatke od korisnika. Od izrazite je važnosti osigurati da su ulazni podaci ispravni, tj. da zadovoljavaju ulazne specifikacije. U nastavku je prikazan odsječak programa napisanog u programskom jeziku Perl u kojem se dohvaćaju podaci o artiklu iz baze podataka prema zadanoj šifri:

```
my $id = $query->param('sifra_artikla');
my $sth = $dbh->prepare(
    "SELECT * FROM artikli WHERE id = $id");
$sth->execute();
```

Sadržaj varijable $id dobiva se iz HTTP GET zahtjeva te se ne provjerava prije izvršavanja SQL upita. Ukoliko je njen sadržaj "`0; DROP DATABASE naziv_baze;`", izvršit će se sljedeći SQL upit: "`SELECT * FROM artikli WHERE id = 0; DROP DATABASE naziv_baze;`", što će za posljedicu imati brisanje cijele baze. Ovakva vrsta napada kod kojeg se modificira sadržaj SQL upita naziva se *SQL injection*. Gornji primjer može se ispraviti na način da se unutar SQL upita ne koriste izravno varijabla, već posebna oznaka za parametar (kod Perlovog DBI sučelja, ta je oznaka znak '?'):

```
my $id = $query->param('sifra_artikla');
my $sth = $dbh->prepare(
    "SELECT * FROM artikli WHERE id = ?");
$sth->execute($id);
```

Sličan problem nastaje ako se prilikom dinamičkog generiranja web stranice ne provjeravaju korisnički uneseni podaci. To je uzrok tzv. *cross-site scripting* (XSS) ranjivosti.

Ovi primjeri pokazuju kako je iznimno važno provjeravati ulazne podatke jer u protivnom napadač može vješto oblikovanim ulaznim podacima uzrokovati rušenje programa, saznati skrivene informacije, promijeniti tijek izvršavanja i sl.

2.2. Race condition propusti

Višekorisnički i višezadaćni sustavi programerima predstavljaju dodatan izazov jer je prilikom razvoja programa potrebno imati na umu i moguće situacije u kojima resursi koje program koristi (datoteke, dijeljena memorija itd.) mogu biti dostupni i drugim procesima koji se istodobno izvršavaju na sustavu. Propust do kojeg dolazi kada jedan proces modificira određeni resurs drugog procesa dok on obavlja neku kritičnu operaciju naziva se *race condition* propust. Sljedeći primjer ilustrira tu pojavu:

```
1  #!/bin/sh
2  echo "/bin/ls" > /tmp/run.$$
3  chmod +x /tmp/run.$$
4  echo "Pokrecem skriptu"
5  sh -c /tmp/run.$$
6  rm /tmp/run.$$
```

Gornja *shell* skripta (niz naredbi koje izvršava ljuska operacijskog sustava) sprema sadržaj "`/bin/ls`" u privremenu datoteku `/tmp/run.$$` (gdje varijabla ljuske `$$` predstavlja oznaku trenutnog procesa), postavi tu datoteku kao izvršnu (eng. executable) pomoću naredbe `chmod` te ju na kraju pokrene. Budući da se taj postupak odvija u nekoliko koraka (nije *atomičan)*, ukoliko napadač izmjeni sadržaj privremene datoteke u nekom trenutku između njenog kreiranja (2. linija) i pokretanja (5. linija), može navesti skriptu na izvršavanje proizvoljnih naredbi.

Primjerice, standardna C biblioteka sadrži funkciju `mkfifo()` koja stvara privremenu datoteku slučajnog naziva. Kako ta funkcija samo kreira novu datoteku unutar datotečnog sustava, a ne *otvara* ju (programer treba eksplicitno pozvati funkciju za otvaranje datoteke ukoliko želi pisati u nju ili je čitati), program koji poziva tu funkciju sadrži *race condition* propust, budući da je moguće izmijeniti sadržaj privremene datoteke u trenutku između njezinog stvaranja pomoću funkcije `mkfifo()` i otvaranja pomoću funkcije `open()`. Problem rješava *atomična* funkcija `mksfifo()` koja nakon stvaranja privremene datoteke istu ujedno i otvara te vraća pozivatelju njezin opisnik (engl. *file descriptor*).

2.3. Preljev međuspremnika

Iako je prvi put uočen još prije nekoliko desetaka godina, *preljev međuspremnika* (eng. buffer overflow) još je uvijek jedan od najčešćih sigurnosnih propusta u programima. Do preljeva dolazi kad se u memorijski spremnik pokuša zapisati više podataka nego što u njega stane, što uzrokuje prepisivanje okolnog dijela memorije. Posljedica toga može biti rušenje programa (ukoliko se primjerice piše izvan granica spremnika u nealocirani memorijski prostor), nepredvidivo ponašanje programa (zbog prepisivanja drugih ključnih podataka), no napadaču je omogućeno i izvršavanje proizvoljnih naredbi.

U sljedećem programu koristi se funkcija `strcpy()` koja ne provjerava granice, pa program sadrži opisani propust. Radi pojašnjenja načina na koji funkcioniraju preljevi međuspremnika, potrebno je prikazati način na koji je program zapisan u memoriji te mehanizam poziva funkcija.

Memorija svakog procesa (programa u izvođenju) može se ugrubo podijeliti na sljedeća tri dijela:

- **tekstualni** segment – sadrži strojne naredbe programa,
- **podatkovni** segment – sadrži inicijalizirane i neinicijalizirane podatke i

- **stogovni** segment – koristi se kod poziva funkcija (za spremanje konteksta, prijenos argumenata, lokalne varijable itd.); stogovni segment raste od viših prema nižim memorijskim adresama (slika 1)

```
#include <stdlib.h>
#include <string.h>
void f(char *str) {
        char buffer[64];
        strcpy(buffer, str);
}
int main(int argc, char *argv[]) {
        if (argc == 1)
                exit(1);
       f(argv[1]);
       return 0;
}
```

Mehanizam poziva funkcije na arhitekturi x86 je sljedeći:

1. Argumenti funkcije stavljaju se na stog (u obrnutom redoslijedu)
2. Strojna naredba CALL sprema trenutnu vrijednost programskog brojila (EIP registar na arhitekturi x86) na stog te nastavlja izvođenje s adrese na kojoj započinje tijelo funkcije
3. Trenutna vrijednost pokazivača na okvir stoga (engl. *frame pointer* ili *base pointer;* EBP registar) sprema se na stog, te se vrijednost pokazivača na vrh stoga (engl. *stack pointer*; ESP registar) kopira u EBP registar; čime je stvoren novi okvir stoga
4. rezervira se dovoljno memorije na stogu za lokalne varijable

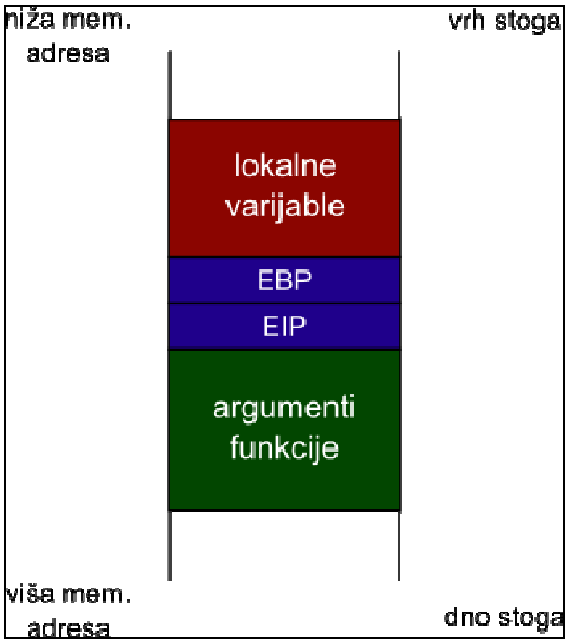
Prilikom izlaska iz funkcije obnavlja se kontekst na sljedeći način:

1. vrijednost EBP registra kopira se u ESP registar; u EBP registar sprema se prethodno sačuvana vrijednost na stogu
2. sa stoga se dohvaća spremljena vrijednost programskog brojila te se izvođenje nastavlja s te adrese

Slika 1 prikazuje izgled stoga nakon poziva funkcije. Uočavamo kako se spremljene vrijednosti EBP i EIP registara nalaze odmah nakon lokalnih varijabli na stogu. Prepisivanjem lokalnog spremnika može se prepisati spremljena vrijednost programskog brojila, što napadač može iskoristiti za preusmjeravanje tijeka programa na proizvoljnu adresu.

Navedeni ranjivi program ponaša se na sljedeći način ako mu se zada preveliki ulazni niz:

```
$ ./bo `perl -e 'print "A"x100'`
Segmentation fault: 11 (core dumped)
$ gdb -q bo bo.core
(no debugging symbols found)...Core was generated by `bo'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.7...(no debugging symbols
found)...done. Loaded symbols for /lib/libc.so.7
Reading symbols from /libexec/ld-elf.so.1...(no debugging
symbols found)...done.
Loaded symbols for /libexec/ld-elf.so.1
#0  0x41414141 in ?? ()
```



Slika 1. Izgled stoga prilikom poziva funkcije.

Kao parametar programu zadan je niz od 100 znakova 'A' (iskorišten je operator x programskog jezika Perl koji ispisuje zadani niz znakova određeni broj puta zaredom), što je više nego što stane u polje `buffer`. Pritom se prepisuje spremljena vrijednost EIP registra. Iz navedenog se ispisa vidi kako je rušenje programa uzrokovano pokušajem izvršavanja naredbe na adresi 0x41414141, što odgovara ulaznom nizu koji je uzrokovao preljev (41 je heksadekadski zapis ASCII znaka 'A'). Slika 2 prikazuje izgled stoga nakon prepisivanja.

Slika 2. Izgled stoga nakon prepisivanja međuspremnika buffer. Spremljeni *stack pointer* (EBP) i povratna vrijednost (EIP) prepisani su s 0x41414141 (četiri znaka 'A').

Stvarni napadi temelje se na prepisivanju spremljene povratne adrese s adresom na koju se želi preusmjeriti izvršavanje programa. Na toj adresi je najčešće umetnut tzv. *shellcode*, niz strojnih naredbi koje pokreću ljusku. U nastavku je dan *shellcode* za operacijski sustav FreeBSD koji pokreće ljusku /bin/sh:

```
char *shellcode =
    "\x31\xc0\x31\xc0\x50\x31\xc0\x50\xb0\x7e\x50\xcd\x80"
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89"
    "\xe3\x50\x54\x53\x50\xb0\x3b\xcd\x80";
```

Danas postoji mnogo načina zaštite od napada omogućenih *buffer overflow* propustima (randomizacija stoga, onemogućavanje izvršavanja instrukcija koje se nalaze u stogovnom segmentu itd.), no isti još uvijek predstavljaju jedan od najčešćih uzroka ranjivosti.

Više informacija o preljevu spremnika može se pročitati u tekstu *Smashing The Stack For Fun And Profit* objavljenom u elektroničkom časopisu Phrack [2].

2.4. *Format string* ranjivosti

Standardna C biblioteka sadrži niz funkcija koje služe za formatiranje ispisa: `printf()`, `fprintf()`, `sprintf()`, `vprintf()` itd. Prvi parametar tih funkcija je tzv. *format string* koji određuje način na koji se ostali podaci ispisuju. Kod poziva funkcije `printf()` izvršava se sljedeće:

- dohvaća se prvi argument (*format string*) s vrha stoga,
- *format string* se analizira te se, ovisno o zastavicama (%d, %s, %p itd.), dohvaća sljedeći argument sa stoga i na odgovarajući način ispisuje.

Ukoliko napadač, pomoću odgovarajućih zastavica može modificirati *format string*, , dobiva mogućnost pregledavanja sadržaja memorije, što mu omogućava rušenje programa ili čak i preusmjeravanje tijeka izvođenja.

U nastavku je prikazan jednostavni ranjivi program:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(int argc, char *argv[]) {
4      if (argc == 1)
5          exit(1);
6      printf(argv[1]);
7      return 0;
8  }
```

Primjer pokretanja naredbe:

```
$ ./fmt '%p %p %p %p'
0x0 0xbfbfeca8 0x28135749 0xbfbfecb0
```

Programu se preko komandne linije predaje niz znakova "%p %p %p %p" koji se u pozivu `printf()` funkcije u liniji 6 koristi kao *format string*. Budući da *format string* sadrži četiri zastavice, dohvaćaju se četiri pokazivača s vrha stoga te se ispisuju adrese na koje pokazuju u heksadekadskom obliku (%p zastavica).

Iskorištavanje *format string* propusta temelji se na korištenju zastavice %n koja uzrokuje zapisivanje broja dosad ispisanih znakova na adresu predanu kao sljedeći argument format funkcije. Uz punjenje (npr. %100d) i %n zastavicu moguće je postići zapisivanje nekog podatka na proizvoljnu adresu. Detaljnije informacije mogu se pronaći u članku *Exploiting Format String Vulnerabilities* čiji je autor *scut* [3].

3. Programi za otkrivanje pogrešaka

Na početku ovog poglavlja opisan je princip rada alata za statičku i dinamičku analizu programa. Nakon toga detaljnije su obrađena tri popularna besplatna alata za statičku analizu s primjerima njihova korištenja.

3.1. Statička analiza

Statička analiza programa je oblik analize koji se provodi nad samim programskim kodom. Početni korak statičke analize je pretvorba programskog koda u odgovarajuću internu reprezentaciju koja je pogodna za daljnju analizu. Taj se postupak obično provodi u nekoliko koraka:

Leksička analiza je postupak kojim se programski kod transformira u niz leksičkih oznaka (eng. *token*), odbacujući pritom nevažne podatke poput praznina i komentara. Slika 3 prikazuje rezultat leksičke analize naredbe `printf("Hello!")`:

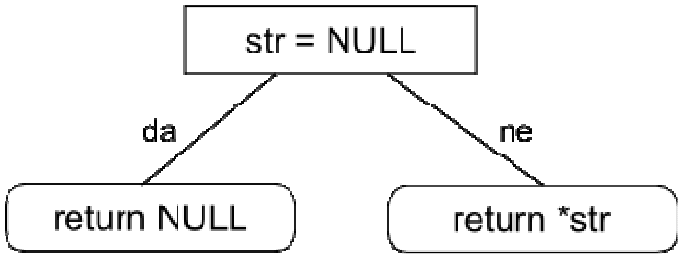
Slika 3. Leksička analiza `printf` naredbe jezika C.

Mnogi alati provode samo leksičku analizu nakon čega traže određene **uzorke**. Npr. ukoliko naiđe na sljedeći niz oznaka koji se dobiva leksičkom analizom naredbe `printf(str)`, takav će alat prijaviti potencijalni *format string* propust:

```
IDENTIFIKATOR(printf) L_ZAGRADA IDENTIFIKATOR(str) D_ZAGRADA TOCKA_ZAREZ
```

Sintaksna analiza je postupak kojim se temeljem leksičkih oznaka te skupa produkcija gramatike određenog jezika gradi stablo parsiranja, odnosno apstraktno sintaksno stablo (eng. *abstract syntax tree)*. Takvo stablo odražava strukturu programskog koda. Izrada apstraktnog sintaksnog stabla omogućava **praćenje kontrolnog toka** programa. Slika 4 prikazuje dijagram toka dobiven analizom sljedećeg programskog odsječka:

```
if (str == NULL)
    return NULL;
else
    return *str;
```



Slika 4. Dijagram toka.

Program koji traži propuste uzrokovane dereferenciranjem *nul*-pokazivača može zaključiti kako u gornjem odsječku takav propust ne postoji. Naime, dereferenciranje pokazivača `str` izvodi se samo u onoj grani kontrolne naredbe **if** za koju vrijedi da pokazivač `str` nije jednak `NULL`.

Alati koji analiziraju tok programa mogu detektirati puno veći broj propusta od onih koji provode samo leksičku analizu. Za više detalja o mehanizmima analize čitatelj se upućuje na [1].

3.2. Dinamička analiza

Dinamička analiza programa je oblik analize koji se provodi **izvođenjem** samih programa. Alati za dinamičku analizu mogu otkriti izvor problema nakon što je do njega došlo. Nedostatak takvog postupka leži u činjenici da je potrebno moći reproducirati problem da bi se mogao naći njegov uzrok, što nije uvijek jednostavan posao.

Područje dinamičke analize je vrlo široko te uključuje alate kao što su *profileri* (alat za analizu performansi pojedinog dijela programa) te memorijski *debuggeri* (alati za detekciju curenja memorije te preljeva međuspremnika). Neki od poznatijih memorijskih debuggera su **Valgrind** koji otkriva memorijske propuste izvodeći program u posebnom virtualnom stroju te **Dmalloc** koji zamjenjuje dio funkcija iz standardne C biblioteke vlastitima, što mu omogućuje detekciju grešaka vezanih uz manipulaciju memorijom.

3.3. Popularni besplatni alati

U ovom dijelu obrađeno je nekoliko popularnih besplatnih alata za *statičku analizu* programa. Prikazani alati namijenjeni su za rad na Unixoidnim sustavima. Svi alati testirani su na operacijskom sustavu FreeBSD 7.2.

3.3.1. Flawfinder

Flawfinder [4] je alat za pronalazak potencijalnih sigurnosnih propusta u izvornom kodu programa pisanih u jezicima C i C++. Flawfinder sadrži bazu funkcija koje su najčešći uzroci sigurnosnih propusta. Pojavljivanje svake od tih funkcija u izvornom kodu prijavljuje se kao mogući propust.

Uz popis nesigurnih funkcija, Flawfinder prikazuje i razinu rizika. Razina rizika je cijeli broj u intervalu [0, 5], a ovisi, osim o samoj funkciji, i o njezinim argumentima. Primjerice, funkcija kojoj je kao parametar predan konstantan znakovni niz smatra se manje rizičnom od funkcije čiji je parametar varijabla.

Flawfinder se koristi zadavanjem opcija te liste datoteka za analizu preko komandne linije. Ukoliko je zadan direktorij umjesto datoteke, Flawfinder analizira sve C/C++ datoteke unutar tog direktorija i svih poddirektorija.

Neke od korisnih opcija:

--inputs	Prikazuje samo one funkcije koje primaju podatke iz okoline.
--falsepositive	Ne prikazuje one rezultate koji su često pogrešno prepoznati kao nesigurni. Ukoliko se koristi ova opcija, funkcije nisu detektirane ukoliko se nakon njihovog naziva ne nalazi znak "(". Time se sprječava pogrešno detektiranje varijabli nazvanih poput funkcija u bazi. Ova opcija nije automatski uključena jer može uzrokovati propuštanje nekih stvarnih propusta.
--minlevel=X	Prikazuje samo one rezultate čija je razina rizika veća ili jednaka X.
--html	Sprema rezultate analize u HTML formatu (umjesto podrazumijevanog tekstualnog ispisa).
--savehitlist=F --loadhitlist=F --diffhitlist=F	Opcije služe za spremanje, učitavanje ili ispis razlika između pronađenih rezultata i onih spremljenih u datoteci F.

Flawfinder prilikom analize ignorira one linije koje sadrže komentar s tekstom "Flawfinder: ignore". Preporučeni obrazac uporabe ovog alata je sljedeći:

- Flawfinder se prvo pokreće nad nizom datoteka te se analiziraju rezultati najviše razine rizika.
- Zatim se koristi opcija --inputs kako bi se pronašli dijelovi programa u kojima se učitavaju podaci te se dodano osigurala validacija istih-
- Nakon analize rezultata, opcionalno se u izvorni kod mogu dodati komentari `/* Flawfinder: ignore */` kako bi se izbjegla lažna upozorenja za one linije koje ne sadrže propuste

Primjer korištenja programa:

```
$ flawfinder buf.c
Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 160
Examining buf.c
buf.c:4:  [5] (buffer) gets:
  Does not check for buffer overflows. Use fgets() instead.
buf.c:3:  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds checking,
  use functions that limit length, or ensure that the size is larger
       than the maximum possible length.

Hits = 2
Lines analyzed = 6 in 0.58 seconds (78 lines/second)
Physical Source Lines of Code (SLOC) = 6
Hits@level = [0]   0 [1]   0 [2]   1 [3]   0 [4]   0 [5]   1
Hits@level+ = [0+]   2 [1+]   2 [2+]   2 [3+]   1 [4+]   1 [5+]   1
Hits/KSLOC@level+ = [0+] 333.333 [1+] 333.333 [2+] 333.333 [3+]
166.667 [4+] 166.667 [5+] 166.667
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
```

U ovom primjeru Flawfinder je detektirao korištenje funkcije gets() koja ne provjerava veličinu spremnika. Dodatno, u trećoj je liniji programa pronašao statičko polje te upozorio na njegovu moguću nesigurnu upotrebu.

Nakon liste rezultata ispisuje se kratki sažetak svih pronađenih ranjivosti. Prikazuje se: broj pronađenih rezultata, ukupan broj analiziranih linija te broj stvarnih linija programskog koda (neprazne linije koje ne sadrže komentare), nakon čega slijedi broj pronađenih ranjivosti po razini rizika. U gornjem primjeru pronađen je jedan rezultat najviše razine (`gets()` funkcija) te rezultat razine 2 (statičko polje). Ispis sažetka se može isključiti pomoću opcije `-D`, odnosno `--dataonly`.

3.3.2. RATS (Rough Auditing Tool for Security)

Poput Flawfindera, i RATS (http://www.fortify.com/security-resources/rats.jsp) također provodi potragu za uzorcima koje smatra potencijalno nesigurnim u izvornom kodu. Ispitivana inačica programa (2.3), ujedno i zadnja u trenutku pisanja dokumenta sadrži ugrađene baze funkcija za jezike C/C++, Perl, PHP, Python i Ruby.

Ispis programa za programski odsječak dan u nastavku je sljedeći:

```
$ rats eval.pl
Entries in perl database: 33
Entries in ruby database: 46
Entries in python database: 62
Entries in c database: 334
Entries in php database: 55

Analyzing eval.pl
eval.pl:5: High: eval
Using user supplied strings anywhere inside of an eval is extremely
dangerous. Unvalidated user input fed into an eval call may allow
the user to execute arbitrary perl code. Avoid ever passing user
supplied strings into eval.

Total lines analyzed: 7
Total time 0.000700 seconds
10000 lines per second
```

Sami programski kod izgleda ovako:

```
#!/usr/bin/perl
use strict;
use warnings;
while (<>) {
      print eval($_), "\n";
}
```

RATS je detektirao korištenje `eval` funkcije koja parsira i izvršava niz Perl naredbi zadanih kao argument funkcije u kontekstu trenutnog programa. Ukoliko može utjecati na argument `eval` funkcije, napadaču je omogućeno izvršavanje proizvoljnih Perl naredbi.

Baza funkcija koje RATS koristi zapisana je u XML formatu što omoguće brzo i jednostavno proširivanje. U nastavku se nalazi primjer jednog takvog zapisa. Za funkciju `strcpy()` navedeno je da uzrokuje preljev međuspremnika (`<BOProblem>...</BOProblem>`), drugi parametar predstavlja izvorni niz kojim se međuspremnik prepisuje, a sama ranjivost okarakterizirana je kao visoko rizična.

Neke od opcija alata RATS:

`--database <datoteka>`	Odabire datoteku koja sadrži bazu ranjivih funkcija.
`--input`	Prikazuje popis korištenih funkcija koje primaju podatke iz okoline.
`--warning <razina>`	Odabire razinu upozorenja. Razina može broj od 1 do 3, gdje 1 (za razliku od Flawfindera) predstavlja propuste najviše razine, a 3 propuste najniže razine rizika. Podrazumijevana razina je 2 (ne prijavljuju se mogući propusti najniže razine rizika).
`--xml` `--html`	Ispiše rezultate u XML, odnosno HTML formatu.

3.3.3. Splint

Splint (http://www.splint.org/), prijašnji LCLint, je alat koji se koristi za statičku analizu programskog koda pisanog u jeziku C. Izdan je pod GPL licencom. Zadnja inačica programa, ujedno i korištena prilikom testiranja ovog alata, je **3.1.2**.

Za razliku od prethodno opisanih programa koji funkcioniraju na način da u programskom kodu samo traže predefinirane uzorke, *splint* radi nešto dublju analizu. *Splint* parsira programski kod te analizira apstraktno sintaksno stablo, za razliku od programa poput Flawfindera koji vrše samo *leksičku* analizu.[5]

Posebnim oznakama (anotacijama) koje se dodaju u kod omogućeno je pronalaženje više različitih programerskih pogrešaka poput memorijskih problema (kao što su dereferenciranje *null* pokazivača, curenje memorije i sl.), korištenja varijabli prije dodjeljivanja vrijednosti, potencijalnih beskonačnih petlji itd.

Kao primjer može se uzeti sljedeći programski odsječak:

```
void f() {
        int *p = malloc(42 * sizeof(int));
        free(p);

        /* neke operacije */

        free(p);
}
```

Funckija `f()`dinamički alocira polje cjelobrojnih vrijednosti, no funkciju `free()` za oslobađanje poziva dvaput, što će vrlo vjerojatno uzrokovati rušenje programa. Splint uočava taj problem:

```
$ splint free.c
Splint 3.1.2 --- 23 Aug 2009

free.c: (in function f)
free.c:7:7: Dead storage p passed as out parameter to free:
               p
  Memory is used after it has been released (either by
  passing as an only param or assigning to an only global).
  (Use -usereleased to inhibit warning)
   free.c:3:7: Storage p released
```

Splint može detektirati i greške do kojih dolazi zbog dereferenciranja nul-pokazivača, no potrebna mu je pomoć od strane programera. Sljedeća funkcija koja vraća prvi znak niza ne provjerava je li njezin argument `NULL`:

```
char prvi_znak (/*@ null @*/ char *str) {
      return *str;
}
```

Uz varijablu `str` nalazi se komentar posebnog oblika `/*@ @*/` koji predstavlja posebnu oznaku splintu. U gornjem slučaju, oznaka `null` daje do znanja splintu da varijabla `str` može poprimiti `NULL` vrijednosti (u protivnom splint podrazumijeva da nijedan pokazivač neće imati vrijednost `NULL`). Splint ispisuje sljedeće upozorenje:

```
$ splint null1.c
Splint 3.1.2 --- 23 Aug 2008

null1.c: (in function prvi_znak)
null1.c:2: Dereference of possibly null pointer str: *str
  A possibly null pointer is dereferenced.  Value is either
  the result of a function which may return null (in which
  case, code should check it is not null), or a global,
  parameter or structure field declared with the null
  qualifier. (Use -nullderef to inhibit warning)
   null1.c:1: Storage str may become null

Finished checking --- 1 code warning
```

Ukoliko se u funkciji `prvi_znak` implementira provjera pokazivača `str`, upozorenje nestaje:

```
char prvi_znak (/*@ null @*/ char *str) {
      if (str == NULL) return '\0';
      return *str;
}
```

```
$ splint null2.c
Splint 3.1.2 --- 23 Aug 2008

Finished checking --- no warnings
```

Dodavanjem posebnih oznaka u programski kod, splint može detektirati i potencijalne preljeve međuspremnika. Uz funkciju `strcpy()` iz standardne C biblioteke navedena je sljedeća oznaka:

```
char *strcpy (char *s1, const char *s2)
/*@requires maxSet(s1) >= maxRead(s2)@*/
```

Pritom `maxSet(s1)` određuje maksimalan indeks `i` takav da se u `s1[i]` može sigurno zapisati vrijednost, dok `maxRead(s2)` određuje maksimalni indeks `j` takav da se iz `s2[j]` može pročitati vrijednost. Ukoliko je zadovoljen uvjet `maxSet(s1) >= maxRead(s2)` neće doći do preljeva prilikom kopiranja znakovnog niza na koji pokazuje varijabla `s2` na adresu na koju pokazuje `s1`.

Sljedeći programski odsječak prikazuje tzv. *off-by-one* ranjivost, posebnu vrstu *buffer overflow* ranjivosti kod koje se u spremnik zapisuje samo jedan bajt više nego što u njega stane:

```
void f(char *str) {
        char buf[64];
        strncpy(buf, str, sizeof(buf)+1);
}
```

Prototip funkcije `strncpy()` je `char *strncpy(char * dst, const char * src, size_t len)` – ona u odredište (`dst`) kopira najviše `len` znakova izvorišta (`src`). U gornjem primjeru je prikazana neispravna upotreba te funkcije – u spremnik `buf` kopira se točno jedan znak više nego što stane u njega.

```
$ splint +bounds obi-wan.c
Splint 3.1.2 --- 23 Aug 2009

obi-wan.c: (in function f)
obi-wan.c:3:2: Likely out-of-bounds store:
    strncpy(buf, str, sizeof((buf)) + 1)
    Unable to resolve constraint:
    requires 63 >= 64
     needed to satisfy precondition:
    requires maxSet(buf @ obi-wan.c:3:10) >= 64
     derived from strncpy precondition: requires maxSet(<parameter
    1>) >= <parameter 3> + -1
  A memory write may write to an address beyond the allocated buffer.
  (Use -likelyboundswrite to inhibit warning)

Finished checking --- 1 code warning
```

Iz ovih se primjera može uočiti kako je splint veoma snažan alat za otkrivanje programskih pogrešaka, no često zahtijeva dodatni posao od programera. Što se više oznaka doda u programski kod, vjerojatnost pronalaska pogrešaka raste.

3.3.4. Dodatni primjeri

U ovom odjeljku nalazi se još nekoliko primjera korištenja alata Flawfinder, RATS i splint.

Splint (s podrazumijevanim postavkama) detektira još jedan oblik memorijskog propusta – vraćanje pokazivača na memoriju alociranu na stogu. Sljedeći programski odsječak ilustrira taj problem:

```
int *f() {
      int x = 42;
      int *p = &x;
      return p;
}
```

Funkcija f vraća pokazivač na lokalnu varijablu čiji životni vijek završava s funkcijom (v. odjeljak 2.3), pa taj pokazivač ostaje „visiti“.

```
$ splint stackref.c
f.c: (in function f)
f.c:4: Stack-allocated storage p reachable from return
          value: p
  A stack reference is pointed to by an external reference
  when the function returns. The stack-allocated storage is
  destroyed after the call, leaving a dangling reference.
  (Use -stackref to inhibit warning)
   f.c:3: Storage p becomes stack-allocated storage
```

RATS uspoređuje vrijeme provjere s vremenom korištenja određenih resursa (npr. datoteka) te može pomoći pri otkrivanju *race condition* propusta. U sljedećem se primjeru koristi `lstat()` funkcija kako bi se ispitalo je li određena datoteka *simbolička veza* (simbolička veza je poseban tip datoteke koja sadrži referencu na neku drugu datoteku; čitanje i pisanje na simboličkoj vezi transparentno se obavljaju nad referenciranom datotekom) . Ako nije, ona se otvara te se u nju nešto zapisuje. Ukoliko napadač uspije modificirati datoteku između poziva `lstat()` i `fopen()`, došlo je do *race conditiona*. Primjer slijedi u nastavku:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
int main(int argc, char *argv[]) {
        struct stat s;
        char *filename;
        FILE *f;

        if (argc < 2)
                exit(1);

        filename = argv[1];
        lstat(filename, &s);
        // nekoliko operacija
        if (! S_ISLNK(s.st_mode)) {
                f = fopen(filename, "w");
                // zapisivanje podataka u datoteku
        }

        return 0;
}
```

Rezultat pokretanja *rats* alata:

```
$ rats --resultsonly lstat.c
lstat.c:13: Medium: lstat
A potential TOCTOU (Time Of Check, Time Of Use) vulnerability exists.
This is the first line where a check has occured.
The following line(s) contain uses that may match up with this check:
18 (fopen)
```

Posljednji primjer u ovom djelu prikazuje detektiranje *format string* propusta u programskom odsječku iz odjeljka 2.4 pomoću Flawfindera:

```
$ flawfinder –D fmt.c
Examining fmt.c
fmt.c:6:  [4] (format) printf:
  If format strings can be influenced by an attacker, they can be
  exploited. Use a constant for the format specification.
```

4. Zaključak

U ovom dokumentu opisane su najčešće programske pogrešake (prvenstveno one vezane uz sigurnost) te načini njihova iskorištavanja. Prikazana su tri alata za detekciju i pomoć u otklanjaju takvih pogrešaka. Iako sasvim jednostavni (većina alata traži samo određene uzorke, primjerice nesigurnu `gets()` funkciju, varijabilni prvi argument `printf()` funkcije i sl.), prikazani alati mogu upozoriti na velik broj potencijalnih sigurnosnih rupa u programskom kodu. Alati poput splinta koji rade dublju analizu programa mogu pronaći još i više ranjivosti, no zahtijevaju veći angažman od programera. Međutim, kao što je i sama zaštita nekog računalnog sustava iznimno kompleksna, niti ovi alati ne mogu dati potpunu sigurnost korisnicima.

Upravo se zbog toga svim korisnicima savjetuje „poštivanje“ svih pravila koje čine neki računalni sustav sigurnim (korištenje antivirusne zaštite, izbjegavanje pokretanja nepoznatih programa, zaštita vlastitih podataka u komunikaciji putem Interneta itd.) te korištenje prethodno opisanih alata za provjeru sigurnosti programskog koda. U svakom slučaju, alati za analizu izvornog koda vrijedan su dodatak arsenalu svakog programera, ali i krajnjeg korisnika jer mogu ukazati na određene sigurnosne nedostatke koji se ne mogu uočiti (dok ne bude prekasno) niti na jedan drugi način.

5. Reference

[1] B. Chess, J. West: *Secure Programming with Static Analysis*, Addison-Wesley, 2007.

[2] Aleph1: *Smashing The Stack For Fun And Profit*, Phrack #49, http://www.phrack.com/issues.html?issue=49&id=14

[3] Scut / team-teso: *Exploiting Format String Vulnerabilities*, http://julianor.tripod.com/bc/formatstring-1.2.pdf

[4] David Wheeler: *Flawfinder*, http://www.dwheeler.com/flawfinder/

[5] Splint Manual, http://www.splint.org/manual/

[6] Common Weakness Enumeration (CWE) – Top 25 Most Dangerous Programming Errors, http://cwe.mitre.org/top25/