SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1766

PISANJE SIGURNIH IZVORNIH TEKSTOVA PROGRAMA

Marin Maržić

Zagreb, lipanj 2011.

Sadržaj

1. Uvod

U radu su opisane smjernice za pisanje sigurnih izvornih tekstova programa te u konačnici za izradu sigurnih programa. Savjeti i smjernice su pisani s naglaskom na gledište programera. Pojam sigurnog programa u ovom radu ima značenje programa koji se izvodi na sigurnosnoj granici, primajući ulaz iz izvora različitih ovlasti od njega samoga. U radu su opisani opće primjenjivi principi i pristupi te su detaljnije razmatrani neki česti problemi. Iako je većina pažnje usmjerena radu u UNIX okruženju (uglavnom Linux) te programskom jeziku C, razmatrani principi gotovo općenito vrijede neovisno o operacijskom sustavu ili programskom jeziku. Zaključno je proveden niz ispitivanja izvornih tekstova programa dostupnih na fakultetskom Webu.

U radu se podrazumijeva općenito poznavanje problema računalne sigurnosti, sigurnosnog modela UNIX okruženja, umrežavanja i programskog jezika C.

U drugom poglavlju se razjašnjavaju neke česte zablude te se odgovara na neka česta i kontroverzna pitanja. U trećem poglavlju se razmatraju sigurnosni zahtjevi sigurnog izvornog teksta programa i vezana norma *Common Criteria*. Četvrto poglavlje se odnosi na dobre sigurnosne principe i pristupe. U petom poglavlju se opisuju načini provjeravanja ulaza programa. U šestom poglavlju je dan pregled metoda izbjegavanja preliva međuspremnika. Sedmo poglavlje se odnosi na pažljivo korištenje vanjskih resursa. U osmom poglavlju se razmatra razumno slanje povratnih informacija, a u devetom poglavlju su opisani problemi specifični pojedinim programskim jezicima. U desetom poglavlju se nalaze analize izvornih tekstova u okviru praktičnog dijela rada.

2. Česta i kontroverzna pitanja

U programerskoj i sigurnosnoj zajednici neka su pitanja vječan izvor rasprava bez jedinstvenog zaključka. Opisane su vrste programa koje radi sigurnosti često zahtijevaju posebnu pažnju. Navedeni su neki razlozi pisanja lošeg, tj. nesigurnog izvornog teksta programa. U ostatku poglavlja se razmatra utjecaj otvorenosti izvornog teksta na sigurnost.

2.1. Koji programi trebaju biti sigurni?

Razni sigurnosni problemi se često rješavaju primjenom istih pristupa i principa. Slijede vrste programa koji trebaju biti napisani tako da zadovoljavaju sigurnosne zahtjeve:

- Programi sa *setuid* ili *setgid* zastavicom. Korisniku pri pokretanju odmah daju ovlasti vlasnika datoteke (*setuid*) ili vlasnikove grupe (*setgid*). Njih je možda najteže osigurati zbog velikog broja mogućih „skrivenih“ ulaza kojima upravlja napadač.
- Preglednici udaljenih sadržaja. Često se koriste za pregledavanje udaljenih podataka poslanih od nepouzdanog izvora. To su programi poput uređivača teksta, Web-preglednika ili preglednika multimedijskih sadržaja. Glavna briga je osiguravanje kako ulazi nepouzdanog izvora ne bi mogli uzrokovati pokretanje proizvoljnih programa. Treba paziti na manje očite ulaze kao npr. inicijalizacijske makroe pokretane pri prezentaciji podataka.
- Aplikacije koje koristi administrator (*root* korisnik). One se ne smiju pouzdati u informacije koje su pod utjecajem korisnika manjih ovlasti.
- Lokalni i mrežno dostupni poslužitelji, tj. demoni (engl. *daemon*).
- Web-aplikacije i CGI (engl. *Common Gateway Interface*) skripte. Pokreću se indirektno putem Web-poslužitelja, ali ovi programi moraju imati snažnu obranu jer poslužitelj ne može spriječiti sve napade na njih.
- Appleti su posebna vrsta programa koji se automatski izvode pri preuzimanju. Zato programer korisnikove applet platforme ne smije vjerovati samom appletu, tj. treba dozvoljavati izvršavanje isključivo sigurnih operacija. Nadalje, programer appleta ne smije vjerovati korisniku i njegovim ulazima [1][2].

2.2. Zašto programeri pišu nesiguran izvorni tekst?

Programeri većinom nemaju namjeru pisati nesiguran izvorni tekst, ali očigledno to ipak čine. Razmatrani su neki od mogućih razloga za takvu situaciju.

U većini škola ne postoji nastavni program koji se zaista bavi računalnom sigurnošću. Knjige i tečajevi programiranja također ne uče sigurnim tehnikama programiranja, zapravo se tek relativno nedavno pojavio broj knjiga posvećenih toj temi. Kad nastavni program posvećen sigurnosti i postoji, on se rijetko bavi pisanjem sigurnih izvornih tekstova programa. Umjesto toga se obično uči o nekim osnovama protokola i kriptografije, a metode formalne verifikacije se rijetko koriste. Na taj se način u potpunosti izbjegava rasprava o iznimno čestim i stvarnim problemima poput preliva međuspremnika, provjeravanja ulaza i formatiranja nizova znakova, a od tih programera će se očekivati pisanje sigurnih programa.

Programski jezik C nije dizajniran kako bi zadovoljio sigurnosne zahtjeve za pisanjem sigurnog izvornog teksta, standardna C biblioteka sadrži funkcije koje su često uzrok sigurnosnih propusta, a jezik je ipak iznimno popularan. Olako i prividno jednostavno korištenje C-a otvara vrata sigurnosnim prijetnjama.

Ne treba podcijeniti ni ljudsku lijenost i nemar kao uzroke problema računalne sigurnosti. Programeri će često svjesno odabrati lakši pristup na uštrb sigurnosti kako bi program ikako proradio, a kad rješenje jednom funkcionira sigurnost ostaje zaboravljena. Većina programera ne razmišlja višekorisnički ni višezadaćno, ne bave se sigurnošću i ne razmišljaju poput napadača. Korisnike je također rijetko briga za sigurnost. Mnogo korisnika ni ne zna za problem, pretpostavljaju da se njima ne može dogoditi ili misle da se situacija ne može promijeniti.

Postoji velika količina starog izvornog teksta kojeg je teško i skupo popravljati. Kako bi se osigurali sigurnosni zahtjevi treba obaviti dodatna ispitivanja i za to utrošiti dodatno razvojno vrijeme [3].

2.3. Je li otvorenost izvornog teksta dobra za sigurnost?

Utjecaj otvorenosti izvornog teksta na sigurnost je jedna od vječnih rasprava među sigurnosnim stručnjacima. Otvoreni programski projekti (engl. *open source*) javnosti izlažu izvorni tekst programa, a stručnjaci se često ne slažu s krajnjim posljedicama te situacije.

Vincent Rijmen, autor pobjedničkog Advanced Encryption Standard (AES) enkripcijskog algoritma, smatra da otvorena priroda Linuxa omogućava lakše uočavanje i popravljanje sigurnosnih propusta:

„Ne samo zato što izvorni tekst više ljudi mogu gledati, već važnije, jer taj model sili ljude da pišu jasniji izvorni tekst i da se pridržavaju normi. To zauzvrat olakšava sigurnosnu analizu.“ [4]

Elias „Aleph1“ Levy je bivši moderator *Bugtraqa*, jedne od najpopularnijih grupa za raspravu o sigurnosti. Sažetak njegovih razmatranja na temu glasi:

„Otvoren programski projekt zasigurno ima potencijal biti sigurnijim od onog kojem izvorni tekstovi nisu javno dostupni. Ali neka ne bude zabune, otvorenost sama po sebi ne garantira sigurnost.“ [5]

Whitfield Diffie je suizumitelj kriptografije javnog ključa, bivši šef sigurnosti u tvrtki Sun Microsystems i trenutni potpredsjednik za informacijsku sigurnost i kriptografiju u ICANN-u. On tvrdi da pristup izvornom tekstu ne znači da će ga programeri pažljivo proučiti, no smatra da se briga može očekivati od programera koji osobno koriste program ili rade za organizaciju koja o njemu ovisi. Whitfield zaključuje:

„Revidiranje programa o kojima poduzeće ovisi je prirodna funkcija informacijsko-sigurnosne organizacije poduzeća. ... Ideja da će korist otvorenog izvornog teksta napadačima prevagnuti nad prednostima za korisnike ide protiv jednog od najbitnijih principa u sigurnosti: tajna koja ne može smjesta biti izmijenjena se treba smatrati ranjivošću. ... Nije realistično da sigurnost računalnog programa ovisi o tajnosti implementacije. Možda uspijete zadržati tajnima detalje rada programa, ali možete li spriječiti obrnut inženjering ozbiljnijeg protivnika? Vjerojatno ne.“ [6]

John Viega, stručnjak računalne sigurnosti, u članku [7] koji se bavi problematikom otvorenosti rezimira:

„Otvoren programski projekt može biti sigurniji nego onaj čiji izvorni tekst nije javno dostupan. Međutim, upravo stvari koje programe otvorenog izvornog teksta čine sigurnim, dostupnost izvornog teksta i činjenica da ga velik broj korisnika može pregledavati i popravljati, također mogu ljude uljuljati u lažan osjećaj sigurnosti.“ [7]

Brian Witten, Carl Landwehr i Michael Caloyannides objavljuju članak gdje zaključuju

kako dostupnost izvornog teksta ide u prilog sigurnosti sustava:

„Kao prvo, pristup izvornom tekstu korisnicima dozvoljava poboljšavanje sigurnosti sustava, ako za to imaju sposobnosti i sredstva. Kao drugo, ograničena ispitivanja upućuju da u nekim slučajevima životni ciklusi otvorenih programskih projekata stvaraju sustave otpornije na greške. Kao treće, ispitivanje provedeno na tri operacijska sustava upućuje da je sustav otvorenog izvornog teksta kroz 12 mjeseci bio pod manjim utjecajem poznatih ranjivosti od ostala dva sustava kojima izvorni tekstovi nisu bili javno dostupni. Konačno, zatvoreni i vlasnički modeli razvoja sustava se suočavaju s manjkom poticaja za podržavanje sigurnijih sustava sve dok su manje sigurni sustavi profitabilniji. Unatoč tim zaključcima, argumenti o ovom važnom pitanju su još u razvoju te su potrebna daljnja mjerenja sigurnosti korisnika.“ [8]

Scott A. Hissam i Daniel Plakosh u "Trust and Vulnerability in Open Source Software" raspravljaju o pozitivnim i negativnim stranama otvorenih programskih projekata. Slično ostalim studijama zaključuju kako je često upitno koliko ljudi pregledava ijedan izvorni tekst. Također demonstriraju kako napadači mogu saznati o ranjivosti u operacijskom sustavu zatvorenog izvornog teksta (Windows) iz zakrpa otvorenog operacijskog sustava (Linux). Programeri operacijskog sustava Linux popravljaju ranjivost prije nego je iskorištena, a napadači pretpostavljaju kako sličan problem još postoji u sustavu Windows. Postojanje programa otvorenog i zatvorenog izvornog teksta iste uloge otvara mogućnost istovremenog otkrivanja ranjivosti oba sustava, a u ovom je slučaju program otvorenog izvornog teksta prije popravljen.

U korist sigurnosti zatvorenih programskih projekata se često navodi kako napadač posjeduje manje informacija pa mu je stoga teže naći ranjivosti. Izvorni tekst je važan za dodavanje novih mogućnosti aplikaciji, no napadači ga ne trebaju kako bi našli ranjivost. Bez obzira na otvorenost izvornog teksta, napadač obično kreće od traženja čestih ranjivosti u programu. Informacije o čestim ranjivostima su javno dostupne i ključne za preventivnu obranu. Obično se program pokreće i pokušavaju mu se zadati problematični ulazni podaci te se promatra odgovor programa u potrazi za tragom ranjivosti. Otvorenost izvornog teksta u takvom pristupu nema utjecaja jer se izvorni tekst ne koristi. Drugi pristup napada uključuje statičku analizu programa, što u slučaju analize zatvorenog programa znači pregled izvršnog teksta dobivenog *disassemblerom* ili teksta programa dobivenog *decompilerom*, a u slučaju otvorenog programa znači pregledavanje izvornog teksta. Iako *decompiler* može vratiti izvršni tekst u nešto slično izvornom tekstu, rezultat je

teško mijenjati i nadograđivati. U procesu *decompiliranja* se gube ljudima razumljivi nazivi varijabli i funkcija, svi komentari i dokumentacija, a izvorni tekst će vjerojatno biti prožet dijelovima u asemblerskom jeziku. Prilikom traženja sigurnosnih ranjivosti to ne znači mnogo jer se u tekstu programa traže određeni uzorci, a nisu potrebni razumljivi nazivi elemenata. To čini *disassembler* i *decompiler* izvrsnim oruđima za traženje načina za napad, a gotovo beskorisnim za popravljanje programa. Zbog moguće nadogradnje i prilagodbe programa, dostupnost izvornog teksta ide u korist obrane, a istovremeno ima malen utjecaj na povećanje ranjivosti sustava.

Nekad se tvrdi kako ranjivost za koju se ne zna ne može biti iskorištena pa je stoga sustav siguran. Nužno je djelovati pod pretpostavkom da netko drugi može pronaći već pronađenu ranjivost. Također, programeri koji su ranjivost otkrili i očuvali tajnom s vremenom mogu na nju zaboraviti. Takve ranjivosti s vremenom ne prestaju biti ranjivostima, već mogu neočekivano uzrokovati probleme i kasnije ih je teško popravljati.

Kad se zatvoren programski projekt naknadno „otvori“, često zbog trenutnog izlaganja ranjivosti postane manje siguran za korisnike, no s vremenom ima potencijal postati mnogo sigurnijim. Oduvijek otvoren programski projekt će biti znatno poboljšan javnim nadzorom prije nego ga počne koristiti veći broj korisnika. To nije pravilo te čin otvaranja izvornog teksta ne garantira poboljšanja sigurnosti programa. Programeri moraju revidirati izvorni tekst, a među njima netko mora znati pisati sigurne programe. Također, jednom kad je problem pronađen, on mora biti brzo popravljen, a zakrpa distribuirana. Osiguravanje pouzdane distribucije sigurnosnih zakrpa krajnjim sustavima je još uvijek problem otvorenih i zatvorenih programa, no situacija se znatno poboljšava širenjem dostupnosti širokopojasnog (engl. *broadband*) Interneta [2][4][5][6][7][8].

3. Sigurnosni zahtjevi

Utvrđivanje razine sigurnosti nekog programa pretpostavlja poznavanje njegovih sigurnosnih zahtjeva. U svrhu normiranog identificiranja, definiranja i evaluiranja sigurnosnih zahtjeva postoji internacionalna norma *The Common Criteria for Information Technology Security Evaluation*, skraćeno zvan *Common Criteria* ili jednostavno CC. CC je norma ISO/IEC 15408 koja služi certifikaciji računalne sigurnosti i rezultat je više desetljeća rada na utvrđivanju sigurnosnih potreba računalnih sustava.

U ovom poglavlju se pregledavaju načini primjene uvedenih koncepata, neformalno se razmatraju ključni sigurnosni zahtjevi te se općenito upoznaje s korištenom terminologijom [9].

3.1. Osnove dokumenta Common Criteria

Common Criteria (skraćeno CC) se sastoji od tri dijela:

- uvoda koji se sastoji od općenitog opisa CC-a,
- sigurnosnih funkcionalnih zahtjeva (engl. *Security Functional Requirements*, skraćeno SFRs) koji se sastoje od opisa raznih sigurnosnih funkcija koje bi mogle biti tražene,
- zahtjeva uvjerenosti u sigurnost (engl. *Security Assurance Requirements*, skraćeno SARs) koji se sastoje od raznih postupaka uvjeravanja u sigurnost proizvoda.

CC se tipično koristi pri stvaranju *Protection Profile* (skraćeno PP) ili *Security Target* (skraćeno ST) dokumenata. U PP-u se identificiraju sigurnosni zahtjevi određenog razreda sigurnosnih proizvoda (npr. pametnih kartica (engl. *smart cards*) ili sigurnosnih stijena (engl. *firewalls*)) te ga obično stvaraju korisniče zajednice. U ST-u se identificiraju sigurnosna svojstva mete evaluacije (engl. *Target of Evaluation*, skraćeno TOE), tj. što proizvod radi, a da to utječe na sigurnost. ST se može referencirati na jedan ili više PP-ova. U slučaju da autori žele uskladiti i evaluirati proizvod s obzirom na PP, PP može služiti kao predložak za ST. Na taj način se korisnici mogu usredotočiti na traženje proizvoda u skladu s PP-ima koji ispunjavaju njihove zahtjeve. Evaluacija PP-a samo osigurava da je

dokument smislen i da zadovoljava razna pravila dokumentacije. Evaluacija ST-a uključuje provjeru dokumentacije, ali i evaluaciju stvarnog sustava (TOE). Svrha evaluacije ST-a je provjeravanje usklađenosti implementacije TOE-a i navedenih ispunjenih sigurnosnih funkcionalnih zahtjeva do specificirane razine uvjerenosti.

Stvaranje PP-a i ST-a zahtijeva identificiranje sigurnosne okoline, odnosno pretpostavki, prijetnji i bitnih organizacijskih politika pod kojima sustav radi. Iz sigurnosne okoline se izvlače sigurnosni ciljevi za proizvod ili vrstu proizvoda. Zatim se odabiru sigurnosni zahtjevi tako da odgovaraju sigurnosnim ciljevima. Iako stvaranje PP-a ili ST-a obično nije jasno i jednostavno kao što je prethodno opisano, korisno je slijediti glavne smjernice postupka kako se ne bi propustile kritične točke.

Funkcionalni zahtjevi opisuju što bi proizvod morao raditi, a zahtjevi uvjerenosti određuju koje mjere će biti poduzete pri provjeravanju implementacije. Većina teksta CC-a se sastoji od popisa definicija normiranih funkcionalnih zahtjeva i zahtjeva uvjerenosti koje bi netko mogao htjeti za svoj proizvod. Autori PP dokumenata biraju zahtjeve koje žele, dok autori ST dokumenata biraju zahtjeve koje njihov proizvod zadovoljava. ST ne mora sadržavati svu sigurnosnu funkcionalnost proizvoda, već samo onu koju treba evaluirati.

Postoje predefinirani skupovi zahtjeva uvjerenosti u sigurnost zvani *Evaluation Assurance Level* (skraćeno EAL), predstavljeni razinama od 1 do 7. EAL-i su korisni kod identificiranja razumnog skupa zahtjeva uvjerenosti u sigurnost. Veća EAL razina nije garancija ni dokaz veće sigurnosti, već mjera snage kojom su provjerene tvrdnje iz ST-a. Proizvodi mogu kombinirati mjere, npr. na EAL 3 mogu biti dodane mjere nedovoljne za postizanje više EAL razine pa se kombinacija zove EAL 3 plus.

Formalna evaluacija akreditiranog laboratorija zahtijeva velik ulog novca, vremena i rada kroz čitav proces, posebno na višim razinama sigurnosti (*Security Assurance Requirements*). Laboratorij će evaluacijom proizvoda utvrditi jesu li sigurnosne tvrdnje istinite. Ako je ustanovljeno da proizvod ne odgovara tvrdnjama, on mora biti popravljen ili tvrdnje moraju biti oslabljene. Svaka tvrdnja mora biti podržana dokazima do specificirane razine uvjerenosti pa jačina i broj tvrdnji te složenost sustava znatno povećavaju cijenu evaluacije. Zbog toga je bitno prije evaluacije ST-a s korisnicima raspraviti o njihovim stvarnim potrebama. Korisnik neće biti zadovoljan evaluiranjem nepotrebno skupih zahtjeva, a ispuštanje ključnih zahtjeva iz ST-a također nije poželjno. Moguće je koristiti PP-ove kao predložak, no bitno je utvrditi je li PP zaista odraz želja korisnika. Korisnik može htjeti evaluirati samo dio zahtjeva PP-a ili kombinaciju PP-ova.

Postojeći PP-ovi mogu biti nedostatni i zbog drukčijeg okruženja ili namjene za proizvod.

Poželjno je provjeriti procese utvrđivanja sigurnosne okoline, ciljeva i zahtjeva jer su bitni u razlučivanju sigurnosnih prioriteta, čak i ako stvaranje PP-a ili ST-a nije u planu. Razmišljanje o pretpostavkama, mogućim prijetnjama i organizacijskim politikama često pomaže u izbjegavanju pogrešaka [1][9].

3.2. Sigurnosno okruženje i ciljevi

Utvrđivanje sigurnosnog okruženja je dobar prvi korak ka definiranju sigurnosnih zahtjeva. Treba razmotriti imaju li napadači pristup hardveru, tj. fizičkom okruženju, utvrditi sredstva koja trebaju zaštitu (npr. baze podataka) i uzeti u obzir svrhu i predviđenu namjenu TOE-a.

Tijekom i nakon utvrđivanja sigurnosnog okruženja dolazi se do prijetnji kojima se sustav ili okruženje moraju suprotstaviti, organizacijskih sigurnosnih politika koje sustav ili okruženje moraju poštovati te skupa pretpostavki (npr. tko je pouzdan). Slijedi određivanje sigurnosnih ciljeva sustava i okoline koji će se suprotstaviti prijetnjama te zadovoljiti pretpostavke i organizacijsku politiku.

U slučaju CC evaluacije bi dalje trebalo odrediti zahtjeve koji udovoljavaju ciljevima i odgovaraju okolini. Ti se zahtjevi određuju biranjem iz popisa mogućih zahtjeva CC-a. Ovakve informacije je dobro pregledati i neformalno zapisati u dokumentaciju čak i ako ne postoji namjera provođenja formalne evaluacije [1][9].

3.3. Sigurnosni funkcionalni zahtjevi

Slijede kratki opisi glavnih razreda funkcionalnih zahtjeva CC-a:

- Sigurnosna revizija (engl. *Security Audit*, skraćeno FAU). Uključuje prepoznavanje, snimanje, spremanje i analiziranje informacija vezanih za sigurnosno relevantne aktivnosti TOE-a (aktivnosti pod kontrolom sigurnosnih funkcija). Proučavanjem zapisa sigurnosne revizije moguće je odrediti koji je korisnik, kada i što radio.
- Komunikacija (engl. *Communication*, skraćeno FCO). Odnosi se na osiguravanje neporecivosti, tj. osiguravanje da izvor ne može poreći slanje, a primatelj ne može poreći primanje poruke.

- Kriptografska podrška (engl. *Cryptographic Support*, skraćeno FCS). Primjenjiva je kad su u TOE-u korištene kriptografske funkcije. Sadrži metode za upravljanje kriptografskim ključevima i njihovo operativno korištenje. Uključuje utvrđivanje operacija i algoritama koji koriste kriptografiju, veličina ključeva i načina ophođenja ključevima.
- Zaštita korisničkih podataka (engl. *User Data Protection*, skraćeno FDP). Sadržani zahtjevi uključuju specificiranje politika kontrole pristupa i toka informacija te razne načine implementiranja tih politika. Razred specificira podršku za *off-line* spremanje, uvoz i izvoz te osiguravanje integriteta kod prijenosa podataka između TOE-ova.
- Identifikacija i ovjera (engl. *Identification and authentication*, skraćeno FIA). Odnosi se na provjeru vjerodostojnosti korisnikovog identiteta. Korisnikove ovlasti za komunikaciju s TOE-om se određuju točnim povezivanjem sigurnosnih atributa (npr. zastavica datotečnog sustava) sa svakim autoriziranim korisnikom. Djelotvornost ostalih razreda zahtjeva, poput sigurnosne revizije i zaštite korisničkih podataka, ovisi o ispravnoj ovjeri korisnika.
- Upravljanje sigurnošću (engl. *Security Management*, skraćeno FMT). Specificira načine upravljanja nekim aspektima sigurnosnih funkcija TOE-a poput sigurnosnih atributa, podataka i funkcija. Uključuje određivanje ovlasti s naglaskom na što veće ograničavanje povjerenja, čak i prema pouzdanijim ulogama.
- Privatnost (engl. *Privacy*, skraćeno FPR). Korisniku pruža zaštitu od otkrivanja i zloupotrebe identiteta. Podržava uvjetnu anonimnost, korištenje pseudonima, nepovezivost i nenadgledivost. Tijekom primjene su mogući snažni konflikti sa zahtjevom neporecivosti.
- Zaštita sigurnosnih funkcija predmeta evaluacije (engl. *Protection of the TSF*, skraćeno FPT). Slično razredu zaštite korisničkih podataka, osim što su u pitanju podaci i funkcionalnost sigurnosnih mehanizama o kojima ovisi zaštita korisničkih podataka. Ako je moguće ugroziti TOE, njene sigurnosne funkcije postaju bezvrijedne pa ona mora imati vlastite mehanizme zaštite. Treba provoditi provjere prije svih pristupa ograničenim akcijama, osigurati fizičku zaštitu TOE, provjeriti niže komponente i periodično raditi provjere čitavog sustava.

- Korištenje resursa (engl. *Resource Utilization*, skraćeno FRU). Sadrži zahtjeve dostupnosti potrebnih resursa poput procesorske snage i podatkovnog prostora. Mora postojati zaštita u slučaju nedostupnosti resursa uzrokovanih kvarom TOE-a. Također treba osigurati točno raspoređivanje važnih i vremenski kritičnih zadataka te onemogućiti monopolizaciju resursa.
- Pristup predmetu evaluacije (engl. *TOE Access*, skraćeno FTA). Sadrži zahtjeve vezane za upravljanje uspostavljanjem korisničke sjednice. Podržava ograničavanje broja mogućih istovremenih sjednica, automatsko zaključavanje ili gašenje sjednica, dopuštanje zaključavanja sjednica od strane korisnika, poruka pri prijavi (npr. datum, vrijeme i lokacija prošle prijave) i uvjetno dozvoljavanje prijave (npr. samo tijekom radnog vremena).
- Pouzdan put/kanali (engl. *Trusted path/channels*, skraćeno FTP). Treba postojati način kojim korisnici mogu biti sigurni da uistinu komuniciraju s „pravim" programom [9].

3.4. Zahtjevi uvjerenja u sigurnost

Slijede kratki opisi nekih mjera CC-a kojima je moguće povećati razinu uvjerenja u sigurnost proizvoda:

- Upravljanje konfiguracijom (engl. *Configuration management*, skraćeno ACM). Mora postojati konzistentno i automatizirano upravljanje verzijama TOE-a s jedinstvenim identifikatorima verzije. To se ne odnosi samo na izvorni tekst, već i na dokumentaciju, izvještaje o problemima, zapise ostalih razvojnih alata i sl.
- Isporuka i rukovanje (engl. *Delivery and operation*, skraćeno ADO). Mehanizam isporuke treba moći spriječiti oponašanje razvojnog programera. Treba pružiti dokumentaciju koja sadrži upute za sigurnu instalaciju, generiranje i pokretanje TOE-a. Također treba stvoriti zapis generiranja TOE-a.
- Razvoj (engl. *Development*, skraćeno ADV). Potrebna je konzistentnost cjelokupne dokumentacije implementacije TOE-a, tj. ne smije biti konfliktnih ili višeznačnih uputa, specifikacija i sl.
- Upute za rukovanje (engl. *Guidance documents*, skraćeno AGD). Moraju biti dostupne upute za rad namijenjene korisnicima i administratorima proizvoda.

Upute trebaju sadržavati smjernice za sigurno rukovanje sustavom te upozorenja na postupke koji mogu ugroziti sigurnost.

- Podrška životnog ciklusa (engl. *Life-cycle support*, skraćeno ALC). Sustavi korišteni u razvoju moraju zadovoljavati sigurnosne zahtjeve (uključujući fizičku sigurnost) i biti pažljivo odabrani. Treba uspostaviti sistematski proces pronalaženja i sanacije nedostataka.
- Ispitivanja (engl. *Tests*, skraćeno ATE). Potrebno je provoditi ispitivanja koja moraju uključivati provjere rada sigurnosnih i općenitih funkcionalnosti.
- Procjena ranjivosti (engl. *Vulnerability assessment*, skraćeno AVA). Treba provoditi analize ranjivosti gdje se netko ponaša kao napadač i pokušava pronaći ranjivosti u proizvodu koristeći dostupne informacije poput dokumentacije te prošlih i javno poznatih ranjivosti. Treba provjeriti korisničke i administratorske upute za korištenje, izbacujući pritom krive, nerazumne i konfliktne smjernice. Moraju postojati upute za sigurnosne procedure u svim načinima rada.
- Održavanje uvjerenja (engl. *Maintenance of assurance*, skraćeno AMA). Treba primjenjivati procedure koje će garantirati da razina uvjerenja u sigurnost neće opasti (npr. svaku promjenu pregledava više ljudi) [9].

4. Sigurnosni principi i pristupi

J.H. Saltzer i M.D. Schroeder su još 1975. godine formulirali deset osnovnih sigurnosnih principa koji do danas ostaju relevantni i primjenjivi u raznim okolnostima:

- Ekonomičnost mehanizma (engl. *Economy of mechanism*). Jednostavnost dizajna mora imati prednost kad je to praktično moguće. Ovaj princip je primjenjiv na bilo koji aspekt sustava i posebno je bitan za sigurnost jer omogućava učinkovito i često provjeravanje izvornog teksta „redak po redak“ te fizičko provjeravanje hardvera.
- Pretpostavljene vrijednosti sigurnosnih postavki (engl. *Fail-safe defaults*). Treba primjenjivati *deny-by-default* politiku, tj. odbijati pristup osim u slučaju eksplicitnog dopuštenja. Primjena ovog principa omogućava da greška u sustavu ne uzrokuje dopuštenje pristupa, već se podrazumijeva sigurno odbijanje.
- Potpun nadzor (engl. *Complete mediation*). Treba provjeravati ovlasti pri svakom pristupu sredstvima sustava. Sistematičnim primjenjivanjem ovog principa stvara se osnova zaštitnog sustava.
- Otvoren dizajn (engl. *Open design*). Dizajn programa ne bi smio biti tajan. Uspješnost sigurnosnog mehanizma ne smije ovisiti o neznanju potencijalnih napadača, već o nekoliko lako zamjenjivih stvari poput lozinki ili ključeva. Takvo razdvajanje mehanizma od ključa omogućava javno proučavanje mehanizama bez brige da će recenzije ugroziti sigurnost.
- Razdvajanje ovlasti (engl. *Separation of privileges*). Dozvola pristupa ne bi smjela ovisiti samo o jednom uvjetu, tj. ključu. Na taj način ugrožavanje jednog sigurnosnog mehanizma neće ugroziti čitav sustav.
- Najmanje ovlasti (engl. *Least privilege*). Treba dodjeljivati samo minimalne nužne ovlasti. Ovaj princip je primjenjiv na svim razinama dizajna, od ovlasti malih dijelova programa do ovlasti korisnika. Njegovom primjenom se ograničava utjecaj grešaka i napada. Također se ograničavanjem mogućih interakcija ovlaštenih programa smanjuje vjerojatnost neželjenih korištenja ovlasti.

- Najmanje dijeljen mehanizam (engl. *Least common mechanism*). Treba minimalno koristiti dijeljena sredstva. Primjenom ovog principa se ograničavaju mogući nesigurni putevi informacija između korisnika.
- Psihološka prihvatljivost (engl. *Psychological acceptability*). Sučelje za ljude bi trebalo biti dizajnirano za jednostavnu i laku uporabu. Na taj se način korisnicima omogućava jednostavna primjena sigurnosnih mehanizama.
- Faktor rada (engl. *Work factor*). Treba usporediti cijenu zaobilaženja obrane sa sredstvima dostupnim napadaču. Ovaj princip je često teško primjenjiv zbog nemogućnosti pouzdanog izračuna potrebnog rada za zaobilaženje većine današnjih sigurnosnih mehanizama.
- Zapisivanje ugrožavanja (engl. *Compromise recording*). Treba osigurati pouzdan trag dokaza ugrožavanja sigurnosti sustava. Ovaj princip je kod računalnih sustava rijetko primjenjiv jer je teško bilo što garantirati kad je sigurnost već ugrožena.

Ovi principi ne predstavljaju apsolutna pravila već služe kao opće smjernice i upozorenja. Ipak, kršenje nekog od tih principa signalizira mogući problem i ne bi trebalo biti olako ignorirano. Svaki princip u nekim situacijama može biti primijenjen, a u nekim ne. Također se neki principi u kombinaciji mogu međusobno suprotstavljati. Programer pri primjeni takvih općenitih principa u obzir mora uzeti njihovu primjenjivost u općem kontekstu rada.

Dobar općenit sigurnosni princip je *defense in depth*, tj. primjena brojnih slojeva obrane kako bi uspješan napad zahtijevao prolaz kroz što više zaštitnih mehanizama [2][10][11].

4.1. Identificiranje ciljeva računalne sigurnosti

Ciljevi računalne sigurnosti su nekad odgovori na poznate prijetnje, a nekad su zakonom propisani i obvezatni. Npr. za banke i financijske ustanove sa sjedištem u SAD-u vrijedi "Gramm-Leach-Bliley" zakon koji nalaže transparentnost dijeljenih osobnih informacija i njihovu sigurnost te ih upućuje da pruže korisnicima mogućnost odbijanja dijeljenja vlastitih podataka.

Ciljevi računalne sigurnosti se često mogu podijeliti na neke općenite ciljeve:

- Povjerljivost (engl. *confidentiality, secrecy*). Samo ovlaštene strane mogu čitati sredstva računalnog sustava. To znači sprečavanje uvida u informacije (nečitljivost)

preuzete presretanjem komunikacije ili čitanjem pohranjenih podataka.

- Integritet (engl. *integrity*). Samo ovlašteni korisnici mogu mijenjati i brisati sredstva računalnog sustava. Informacije treba zadržati u nepromijenjenom obliku tijekom prijenosa i pohrane.
- Raspoloživost (engl. *availability*). Sredstvima računalnog sustava ovlašteni korisnici mogu pristupati s razumnim vremenskim kašnjenjem. Informacije moraju biti dostupne, a sustavi i usluge u operativnom stanju usprkos mogućim nepredvidljivim ili neočekivanim događajima.

Ciljevi se različito grupiraju ovisno o literaturi. Negdje se kao cilj posebno identificira neporicanje (engl. *non-repudiation*), tj. sposobnost dokazivanja kako je pošiljatelj poslao ili primatelj primio poruku unatoč kasnijem pokušaju poricanja događaja. Privatnost (engl. *privacy*) se nekad navodi odvojeno od povjerljivosti te se definira kao zaštita povjerljivosti identiteta korisnika. Većina ciljeva uključuju identifikaciju (engl. *identification*) i ovjeru (engl. *authentication*) pa se i oni ponekad navode kao zasebni ciljevi. Kao bitniji ciljevi se često navode i odgovornost (engl. *accountability, auditing*) te kontrola pristupa (engl. *access control*). Odgovornost se definira kao potpuna odgovornost korisnika za vlastite akcije i postiže se stalnim nadgledanjem aktivnosti sustava. Kontrola pristupa se definira kao ograničavanje pristupa informacijama i uslugama čime se ograničavaju i akcije nad tim sredstvima.

Bez obzira na podjelu ili grupaciju ciljeva važno je moći identificirati sigurnosne ciljeve vlastitog programa kako bi prijetnje tim ciljevima mogle biti pravovremeno prepoznate [1][12][13].

4.2. Sigurnosni zahtjevi u primjeni

Često postoji sukob između sigurnosti i drugih općih inženjerskih praksa i principa poput lakoće uporabe. Npr. korištenje sigurnije veze zahtijeva dodatnu ovjeru ili možda sigurna instalacija zahtjeva dodatnu konfiguraciju. Takvi „prividni“ problemi često mogu biti riješeni preinakom sustava sigurnosti bez kompromisa lakoće korištenja. Nekad postoji i sukob između sigurnosti i apstrakcije, npr. neke programske biblioteke visoke razine mogu imati sigurne ili nesigurne implementacije, ali o tome nema riječi u specifikaciji. U tom slučaju treba samostalno implementirati siguran program, iako je možda biblioteka ta koja bi trebala biti popravljena [1][10][11].

4.3. Paranoja kao vrlina

Programi koji ne trebaju biti sigurni imaju mnoge nedostatke (engl. *bugs*) koji su obično izazvani rijetkim okolnostima, a često slučajno otkriveni normalnim korištenjem. Korisnik u tom slučaju može izbjegavati okolnosti u kojima se javlja kvar ili prestati koristiti program.

S programima koji trebaju biti sigurni je situacija obrnuta, a izravne posljedice previda programera su značajnije. Korisnik sada namjerno pokušava izazvati kvar, odlučno tražeći rubne, rijetke i nevjerojatne uvjete, u nadi da će tako ostvariti neovlašten pristup sustavu. Zbog toga programer ima potrebu za drukčijim i mnogo opreznijim razmišljanjem. Općenita paranoja i razmišljanje poput napadača su pri sigurnom programiranju potrebni zbog težih posljedica neuspjeha sigurnosti, tj. kvarova sigurnih programa [1][2].

5. Provjeravanje ulaza

Neki ulazi programa pristižu od nepouzdanih izvora pa moraju biti provjereni ili filtrirani. Dobra je praksa identificirati neke nelegalne ulaze u svrhu osnovne provjere zdravog razuma i ispravnosti programa, no takav pristup nije dovoljan. Treba primijeniti *deny-by-default* politiku, tj. odrediti što je ispravan ulaz i samo takve propuštati.

Ako je provjera ispravnosti neispravna, može doći do nekonzistentnosti između programa koji raspolažu istim ulazom, a mogući sigurnosni propusti mogu ostati nepokriveni. Zbog toga je provjere dobro držati organizirane na jednom mjestu za kasnije revizije. Također je bitno odrediti kojim ulazima nepouzdan korisnik upravlja, jer se samo za takve ulaze nužno brinuti [1][2].

5.1. Česte vrste podataka i provjera

U radu s poljima (engl. *arrays*) javljaju se vrlo česte greške „*off by one*“, tj. pristupa se za jednu lokaciju izvan rezerviranog spremničkog prostora. Prilikom izračunavanja vrijednosti indeksa polja treba naročito paziti jer se zbog malog odstupanja takvi kvarovi ne moraju odmah ili ikad jasno pokazati u radu, a uzrok su sigurnosnih ranjivosti.

Znakovni nizovi (engl. *strings*) mogu sadržavati svakakve podatke pa treba identificirati dopuštene znakove i uzorke (npr. regularnim izrazom) te odbaciti podatke koji dopuštenom uzorku ne odgovaraju. Prilikom korištenja znakovnih nizova koji sadrže upravljačke znakove (engl. *control characters*) ili specijalne znakove (engl. *metacharacters*) javljaju se posebni problemi pa je takve znakove poželjno odmah prekinuti (engl. *escape*). Takvi problemi se javljaju i kod aplikacijski specifičnih formata spremanja podataka (npr. graničnici (engl. *delimiters*), navodnici, znak '<'). Najbolje je iskoristiti prekidni niz (engl. *escape sequence*) specifičnog formata ako takav postoji ili takve podatke filtrirati odmah na ulazu. Općenit problem je prezentacija znakovnog niza drugim korisnicima gdje ovisno o kontekstu on može poprimati posebna značenja (npr. HTML oznake u porukama koje će na Web-stranici biti objavljene drugim korisnicima).

Brojeve treba ograničiti na odgovarajuće minimalne i maksimalne dopuštene vrijednosti. Također treba paziti na tip varijable u kojoj je broj spremljen s obzirom na predznak. Npr.

C *char* u biti predstavlja cijeli broj s predznakom (engl. *signed*) pa vrijednost pohranom u *integer* ostaje negativna što možda nije očekivano ponašanje.

E-mail adrese je zbog podrške starijih formata te kompliciranih i specifičnih pravila specifikacije teško sveobuhvatno provjeriti po specifikaciji. Takvi pokušaji rezultiraju ogromnim regularnim izrazima pa se obično koristi neki restriktivniji, proizvoljan i jednostavno provjerljiv podskup pravila koji obuhvaća česte oblike e-mail adresa.

Prilikom upravljanja URI-jem (engl. *Uniform Resource Identifier*, skraćeno URI) prvo treba dekodirati sva kodiranja (URL i UTF-8 kodiranja) da nešto poput '..' ne bi promaklo provjerama. Programi koji URI-jem dohvaćaju lokalne podatke trebaju paziti kako ne bi bilo moguće izaći iz korijenskog direktorija unutar kojeg poslužitelj radi. Najčešći načini bježanja iz korijenskog direktorija uključuju korištenje simboličkih poveznica (engl. *symbolic links*) ili '..' pa je takve upite nužno filtrirati. Ako program koristi URI unutar drugih podataka, treba osigurati da nepouzdani korisnici ne mogu ubacivati URI-je štetne drugim korisnicima [1][2][14][15][16].

5.2. Ulazi naredbenog retka

Mnogi programi ulaz primaju iz naredbenog retka (engl. *command line*). Nepouzdan korisnik može upravljati ulazom naredbenog retka nekog *setuid/setgid* programa. Napadač kroz naredbeni redak pozvanom programu može predati razne podatke (npr. kroz `exec(3)` familiju funkcija). Programi ne smiju vjerovati ni nazivu programa iz nultog argumenta jer se on također može proizvoljno postaviti. Sigurni programi trebaju u potpunosti provjeriti ulaz s naredbenog retka kako bi se obranili od zlonamjernih vrijednosti [1][2][17].

```
#include <unistd.h>
// postavljanje proizvoljnih argumenata i poziv programa
char *cmd[] = { "ping", "-v", (char *) 0 };
execv("/bin/ping", cmd);
```

5.3. Varijable okruženja

Varijable okruženja su interno pohranjene kao pokazivač na polje pokazivača na nizove znakova, a polje je završeno (engl. *terminated*) pokazivačem `NULL`. Nizovi znakova su oblika `'NAZIV=vrijednost'` i završeni su znakom `null`.

```
#include <unistd.h>
extern char **environ; // pokazivač varijabla okruženja
```

Kad program pozove drugi program, on varijable okruženja (engl. *environment variables*) može postaviti na proizvoljne vrijednosti. Na taj način napadač u potpunosti može upravljati okruženjem *setuid/setgid* programa. Neke su varijable okruženja opasne zbog programa i biblioteka koji ih koriste na nedokumentirane ili opskurne načine. Način interne pohrane varijabli okruženja je još jedan od uzroka njihove opasnosti. Moguće je unijeti više varijabli istog naziva pa neki program može provjeriti jednu vrijednost, a zapravo koristiti drugu. U slučaju Linuxa, GNU C bibliotekama se programe pokušava zaštititi od takvog ponašanja, no neki programi direktno pristupaju varijablama okruženja pa u tom slučaju nije jasno koja će vrijednost biti odabrana.

```
#include <unistd.h>
// postavljanje proizvoljnog okruženja i poziv programa s
argumentima
char *env[] = { "HOME=/home/myuser", "USER=myuser", (char *) 0 };
execle("/bin/ls", "ls", "-al", (char *) 0, env);
```

Jedino pravo rješenje je odabir potrebnih varijabli okruženja i odbacivanje ostatka. Za sigurne programe treba pažljivo izvući kratak skup nužnih varijabli okruženja nakon čega bi čitavo okruženje trebalo pobrisati i onda sigurno postaviti vrijednosti potrebne za rad. Jednostavan način za brisanje okruženja u C/C++ je postavljanje globalne varijable `environ` (definirana u `<unistd.h>`) na `NULL`. Ako se programira u ljusci (engl. *shell*), moguće je koristiti `'/usr/bin/env'` program s opcijom `'-'`, nizom naziva i pripadajućih vrijednosti oblika `'naziv=vrijednost'` te nazivom programa i njegovim argumentima. Program je dobro pozvati s punom stazom jer varijabla okruženja PATH može biti postavljena na opasnu vrijednost. Prilikom korištenja programskih jezika koji ne dopuštaju direktnu manipulaciju varijablama okruženja, moguće je stvoriti program omotač (engl. *wrapper*) koji će varijable sigurno postaviti te pozvati pravi program.

Ako je korisnicima dopušteno postavljanje varijabli okruženja, oni će vjerojatno moći izaći iz ograničenih korisničkih računa. Mijenjanje okruženja je također moguće kroz promjenu datoteka u vlastitom `home` direktoriju, npr. `.login` ili `.ssh/environment` u slučaju OpenSSH-a, ili kroz protokole koji podržavaju prijenos varijabli okruženja (npr. telnet). Ograničeni korisnički računi ne bi smjeli dozvoljavati promjenu ni dodavanje datoteka izravno u `home` direktoriju [1][2][17][18][19].

5.4. Opisnici datoteka

Program prima skup već otvorenih opisnika datoteka (engl. *file descriptors*) što *setuid/setgid* programi moraju uzeti u obzir. Iskorištavanjem tog ponašanja napadač može uzrokovati razne neočekivane situacije. Npr. ako napadač zatvori standardni izlaz, sav će standardni izlaz biti poslan u prvu otvorenu datoteku. Ne smije se pretpostaviti da će nova datoteka biti otvorena s predodređenim identifikacijskim brojem opisnika ni da će se uopće otvoriti. Također se ne smije pretpostaviti da standardni ulaz (`stdin`), standardni izlaz (`stdout`) i standardni izlaz za greške (`stderr`) pokazuju na terminal ni da su uopće otvoreni. Neke C biblioteke automatski otvaraju `stdin`, `stdout` i `stderr` ako već nisu otvoreni, ali to nije tako u svim UNIX okruženjima. Također, biblioteke nisu uvijek potpuno pouzdane, npr. na nekim sustavima je moguće stvoriti *race condition* koji će navedeno automatsko otvaranje onemogućiti, a program će se ipak pokrenuti [1][17].

5.5. Nazivi datoteka

U UNIX okruženju su često dopušteni nazivi datoteka koji se sastoje od gotovo bilo kakvih vrijednosti. Datoteke neočekivanih naziva mogu omogućiti ugrožavanje sustava. Nazivi datoteka koji mogu uzrokovati probleme uključuju:

- Nazivi datoteka s vodećim crticama ('-'). Ako je ovakav naziv predan drugom programu kao parametar, mogao bi biti interpretiran kao opcija. Zato pri pozivanju drugih programa prije parametra naziva datoteke treba ubaciti '--' (kako bi se prekinulo procesiranje opcija) ili promijeniti naziv datoteke (npr. ubacivanjem `./` ispred naziva kako crtica ne bi bila vodeći znak).
- Nazivi datoteka s upravljačkim znakovima. Znakovi novog retka (engl. *newline*) i kraja retka (engl. *carriage return*) se često interpretiraju kao graničnici argumenata. Oni također mogu neželjeno lomiti unose u zapisnik (engl. *log entries*) u više redaka. Znak ESCAPE u nazivu također može smetati emulatoru terminala, uzrokujući neželjene akcije izvan kontrole korisnika.
- Nazivi datoteka s razmacima. Ovakvi nazivi ponekad mogu biti interpretirani kao više argumenata što stvara probleme. U ljusci treba koristiti dvostruke navodnike oko svih naziva datoteka. Također je preporučljivo zabraniti vodeće i krajnje

razmake u nazivima datoteka jer su tada najmanje vidljivi i mogu zbuniti ljudske korisnike.

- Pogrešno kodirani nazivi datoteka. Npr. program može naziv datoteke dekodirati kao valjan UTF-8 naziv, a naziv može imati nevaljano dugo UTF-8 kodiranje koje se može različito interpretirati.
- Nazivi datoteka sa znakovima posebnog značenja aplikaciji. Znakovi mogu imati posebno značenje internim oblicima pohrane podataka neke aplikacije (npr. znakovi `"`, `<`, `;`)

Ako se nepouzdan ulaz interpretira kao naziv datoteke, dobro je zabraniti znakove koji uzrokuju promjenu direktorija ('. .', '/'). Također je dobro isključiti podršku za *globbing*, tj. širenje naziva datoteka uz pomoć znakova `*`, `?`, `[ ]` i `{ }`. Ako *globbing* nije potreban, poželjno je koristiti funkcije koje ga ne primjenjuju (npr. `fopen(3)`) ili ga onemogućiti prekidanjem *globbing* znakova u ljusci. Ako je *globbing* neophodan, potrebno ga je oprezno implementirati. Složen *globbing* može dugo trajati i efektivno izvesti uskraćivanje usluga (engl. *denial-of-service*, skraćeno DoS) na čitavom računalu. Moguće rješenje je pokretanje takvih programa u procesima s ograničenim pristupom resursima računala [1][19].

5.6. Sadržaj datoteka

U sigurnom programu se ne smije koristiti sadržaj datoteke koji mogu mijenjati nepouzdani korisnici. Ako korisnik smije mijenjati datoteku, njen direktorij ili ijedan viši direktorij, to obično znači da je sadržaj datoteke nepouzdan. Ako program ipak mora čitati sadržaj nepouzdane datoteke, takav ulaz treba provjeriti strogo kao i ostale ulaze [1].

5.7. Ulazi Web-aplikacija

Aplikacije temeljene na Webu (engl. *web-based applications*) ili Web-aplikacije (engl. *web applications*) poput CGI skripata se pokreću na pouzdanom serveru, ali ulazni podaci stižu kroz općenito nepouzdan Internet pa se moraju provjeravati. Provjeravanje moraju činiti poslužitelji jer su korisnici općenito nepouzdani.

CGI skripte su česte mete napada uključivanjem specijalnih znakova u njihov ulaz. Dodatan problem se javlja zbog postotnog kodiranja znakova (engl. *percent-encoding*,

URL encoding) u CGI ulazu. CGI biblioteka ili programer moraju takav ulaz točno dekodirati i onda provjeriti valjanost ulaza. Česte problematične vrijednosti koje također moraju biti pravilno obrađene uključuju `%00` (`null` znak) i `%0A` (znak za novi redak). Također je bitno dekodirati samo jednom jer bi inače ulaz poput `%2500` u prvom prolazu mogao biti dekodiran kao `%25` u %, a u drugom pogrešno `%00` u `null` znak.

Web-aplikacije imaju pristup kolačićima koji se moraju smatrati nepouzdanim jer mogu biti proizvoljno postavljeni i poslani. Kolačići mogu biti korišteni i za praćenje korisnika pa ih mnogi korisnici onemogućuju. Stoga bi Web-aplikacija, ako je ikako moguće, trebala biti dizajnirana da ne ovisi o kolačićima. Također je poželjno ograničiti korištenje trajnih kolačića jer omogućuju laku zlouporabu.

Neke Web-aplikacije podatke nepouzdanog korisnika (napadača) prosljeđuju do aplikacije drugog korisnika (žrtve). Žrtvina aplikacija (npr. Web-preglednik) takve podatke može obraditi na način štetan žrtvi. U slučaju Web-aplikacija koje koriste HTML ili XML oznake taj se problem zove *cross-site scripting (XSS)* ili *cross-site malicious content*. Problem se rješava filtriranjem (izbacivanje problematičnih znakova), kodiranjem (transformiranje problematičnih u bezopasne znakove) ili provjeravanjem (osiguravanjem da su samo sigurni ulazi propušteni) ulaza korisnika. CERT preporučuje filtriranje i kodiranje podataka na izlazu, ali ovisno o situaciji je to smislenije učiniti odmah na ulazu. Neovisno o metodi, bitno je u obzir uzeti sve izlaze i ulaze što predstavlja glavni problem. Ovakve se obrambene metode mogu zaobići ako nije moguće upravljanje kodiranjem kojim će izlaz sigurnog programa biti interpretiran. Rješenje je u slučaju HTML-a lagano jer jednostavno treba postaviti željenu vrijednost `charset` zaglavlja dokumenta. Kodiranje je parametrom `charset` moguće postaviti i direktno u izlazu protokola HTTP [1][20].

```
<meta http-equiv="Content-Type" content="text/html"; charset=UTF-8">
```

5.8. Ostali ulazi

Nužno je upravljanje svim ulazima, što je kod *setuid/setgid* programa posebno teško zbog velikog broja mogućih ulaza. Npr. program mora u obzir uzeti trenutni direktorij koji je pri pokretanju programa poželjno promijeniti na valjan direktorij punog naziva (npr. pomoću `chdir(2)`). Među ostalim ulazima koje programi moraju uzeti u obzir su: trenutni direktorij, signali, mapiranja memorije (engl. *memory maps*), komunikacija između

procesa, brojači u tijeku, ograničenja resursa, prioritet raspoređivanja (engl. *scheduling priority*) i *umask* [1][2].

6. Izbjegavanje preliva međuspremnika

Preliv međuspremnika (engl. *buffer overflow*) je problem programa koji piše preko granica međuspremnika te prepiše susjednu memoriju. Takva se situacija može dogoditi učitavanjem ulaza korisnika u međuspremnik, ali i kod mnogih drugih vrsta procesiranja u programu.

Preliv međuspremnika je uzrok mnogih ranjivosti i često je opisivan kao najštetniji problem računalne sigurnosti. Neformalna anketa na Bugtraq popisu adresa je 1999. utvrdila kako su dvije trećine sudionika smatrale da je najveći uzrok sigurnosne ranjivosti sustava preliv međuspremnika. Od 1997. do ožujka 2002. pola sigurnosnih upozorenja CERT/CC-a su bila vezana uz ranjivosti preliva međuspremnika. Prelivi međuspremnika imaju i dugu povijest omogućavanja izvođenja proizvoljnih programa na igraćim konzolama. 2003. godine je preliv međuspremnika u licenciranim Xbox igrama omogućio izvođenje nelicenciranog softvera, „PS2 Independence Exploit“ je isto omogućio na PlayStation 2, a „Twilight hack“ je to omogućio na igraćoj konzoli Wii.

U gotovo svim višim programskim jezicima/okruženjima (npr. Python, PHP, Perl, Ruby, Java, .NET, Ada, Lisp itd.) preliv međuspremnika je nemoguć zbog automatskog širenja polja ili nekog oblika ugrađene detekcije i prevencije. C i C++ su iznimke jer ne pružaju ugrađenu zaštitu protiv pisanja izvan granica međuspremnika i stoga su ranjivi na ovakav napad. Taj se problem zato često asocira s jezicima C i C++ koji su među najpopularnijim programskim jezicima.

Metode iskorištavanja ranjivosti variraju ovisno o arhitekturi i operacijskom sustavu, no ako program dopušta preliv spremnika on može biti iskorišten. U slučaju lokalne varijable međuspremnika je prelivom moguće izmijeniti vrijednosti ostalih lokalnih varijabli ili funkciju natjerati da izvede proizvoljan tekst programa. Preliv međuspremnika na gomili (engl. *heap*) može omogućiti promjenu ostalih varijabli programa. Mogućnost preliva međuspremnika, neovisno o klasifikaciji, metodi iskorištavanja ili računalnom sustavu, predstavlja veliki problem sigurnosti.

Izbjegavanje preliva međuspremnika se uglavnom svodi na pažljivo i točno pisanje izvornog teksta koji se bavi upravljanjem međuspremnicima, koristeći pritom sigurne biblioteke. Prije svakog pisanja u međuspremnik program treba provjeriti da je dostupno

dovoljno prostora ili na neki drugi način mora osigurati da se preliv neće dogoditi. Naravno, dosadašnje iskustvo pokazuje da je to lakše reći nego učiniti [1][2][21][22][23].

6.1. C/C++ opasnosti

Standardna C biblioteka sadrži neke funkcije koje samostalno ne provjeravaju granice pri pristupu međuspremnicima pa mogu biti opasne ako programer ne osigura da granice neće biti pređene. Korištenje takvih funkcija bi općenito trebalo izbjegavati. Opasne funkcije uključuju `strcpy(3)`, `strcat(3)`, `sprintf(3)`, `vsprintf(3)` i `gets(3)` umjesto kojih je preporučeno korištenje `strncpy(3)`, `strncat(3)`, `snprintf(3)` i `fgets(3)`. Funkcija `snprintf(3)` bi trebala biti relativno sigurna, što je slučaj u modernim UNIX okruženjima, ali u nekim starijim Windows i UNIX okruženjima nije pravilno implementirana. Funkcije `scanf(3)`, `fscanf(3)`, `sscanf(3)`, `vscanf(3)`, `vsscanf(3)` i `vfscanf(3)` su također opasne i ne bi smjele biti korištene bez upravljanja maksimalnom duljinom ulaznog niza (glavni problem je format `%s`). Ostale potencijalno opasne funkcije uključuju `realpath(3)`, `getopt(3)`, `getpass(3)`, `streadd(3)`, `strecpy(3)`, `strtrns(3)`, `getwd(3)` i `select(2)`. Npr. prilikom poziva `getwd(3)` treba osigurati da je međuspremnik velik barem `PATH_MAX` okteta, a kod poziva pomoćnih makroa funkcije `select(2)` (`FD_SET()`, `FD_CLR()` i `FD_ISSET()`) treba provjeriti da je indeks `fd` unutar granica (0 <= `fd` <= `FD_SETSIZE`).

Osim funkcija koje se bave nizovima znakova, problemi su mogući i kod manipulacije brojevima. Zbog slabog provjeravanja tipova cjelobrojnih varijabli C-a, programer ih mora pažljivo koristiti. Napadač može predati negativan ili dovoljno velik broj koji se kod provjere prevelikih vrijednosti interpretira kao broj s predznakom, a kod pristupa memoriji kao broj bez predznaka (engl. *unsigned*).

```
/* Primjer preliva međuspremnika zbog predznaka tipa varijable */

char buf[MAXBUF];
int len, fd;

scanf("%d", &len); // učitan negativan ili prevelik len

if (len > MAXBUF) { // nema provjere za len < 0
        ... // greška, tražen prevelik len
}

// read prima size_t koji je unsigned tip, događa se cast varijable
// len iz int u size_t -> preliv međuspremnika zbog prevelikog len
read(fd, buf, len);
```

Nekad preliv međuspremnika može biti uzrokovan i skraćivanjem (engl. *truncation*) cjelobrojnih vrijednosti kod pretvorbe iz većeg u manji tip (npr. 64bit `size_t` u 32bit `uint32_t`) ili prelivima cjelobrojnih varijabli (engl. *integer overflow*).

```
/* Primjer preliva međuspremnika zbog preliva integera */

char *buf;
size_t len;

scanf("%d", &len); // učitan len maksimalne size_t vrijednosti

// nema provjere veličine

buf = malloc(len + 1); // +1 za NUL znak, ali može preliti broj u 0
read(fd, buf, len);    // buf premal, događa se preliv međuspremnika
buf[len] = '\0';
```

U tim je slučajevima preliv međuspremnika moguć zbog dodjeljivanja nedovoljne količine memorije [1][17][23].

6.2. C/C++ rješenja

Iako većina viših programskih jezika imaju ugrađene mehanizme koji onemogućuju preliv međuspremnika, programeri se zbog raznih razloga ipak odlučuju na rad s jezicima C i C++.

Postoji mnogo mogućih protumjera za preliv međuspremnika. Većina razlika u pristupu proizlazi iz podjele na statički dodijeljene (fiksne veličine) i na dinamički širene međuspremnike koji svaki imaju vlastite prednosti i nedostatke. Čak i uz sve protumjere ipak je neizbježna potreba za iznimno opreznim programiranjem u jezicima C i C++ [1][23].

6.2.1. Statički ili dinamički dodjeljivani međuspremnici

Spremnici imaju ograničenu količinu prostora pa postoje dvije glavne mogućnosti za rješavanje nedostatka prostora. U slučaju statički dodijeljenog međuspremnika popunjavanje početno dodijeljenog prostora znači kraj dodavanja. Prilikom dinamičkog pristupa međuspremnik se automatski širi sve dok sustav ne ostane bez dostupne memorije. Oba pristupa imaju različite sigurnosne posljedice.

Statički dodijeljen međuspremnik ima fiksnu veličinu što može biti iskorišteno. Višak podataka se odbacuje, no moguće je da se ostatak koristi. Napadač može namjestiti tako

dug niz znakova da nakon skraćivanja u međuspremniku ostane ono što je on htio. U korištenju statičkog pristupa bitno je posvetiti pažnju duljini izvora i odredišta, a korisno je dodati provjere konačnog rezultata.

Dinamičko dodjeljivanje spremnika je preporučeno GNU programerskim smjernicama jer omogućuje korištenje proizvoljno dugih ulaza. Glavni problem takvog pristupa je ostajanje bez memorije. Moguće je ostati bez memorije i u trenutku izvođenja nekog drugog dijela programa, a ne samo kad je preliv međuspremnika očekivan, što stvara dodatne probleme. Također je zbog povremene neefikasnosti dinamičkog dodjeljivanja memorije moguće ostati bez prostora, iako u biti postoji dovoljno virtualne memorije za nastavak izvođenja. Trošenje većine prostora izaziva prebacivanje podataka između diska i memorije što računalo čini gotovo beskorisnim pa je moguć napad uskraćivanjem usluga. Zbog toga ipak treba ugraditi neka razumna ograničenja i dizajnirati program tako da sigurno upravlja nedostatkom memorije [1][23].

6.2.2. Standardna C biblioteka

Korištenje relativno sigurnih funkcija standardne C biblioteke poput `strncpy(3)` i `strncat(3)` je jednostavan i prenosiv (engl. *portable*) pristup rješavanju problema kod statički dodijeljenih međuspremnika. Ove funkcije su općenito dostupne, svaki programer ih poznaje i općenito zadovoljavajuće služe svrsi. Ako je odabran ovakav pristup, treba se dobro upoznati sa značenjem i radom navedenih funkcija jer se mogu pogrešno primijeniti.

```
#include <string.h>
char *strncpy(char *dest, char *src, size_t n);
char *strncat(char *dest, char *src, size_t n);
```

Obje funkcije kao argument primaju količinu preostalog prostora u međuspremniku koja se svakim dodavanjem i brisanjem podataka mijenja. Zbog toga programeri moraju stalno računati količinu preostalog prostora, što je lagano pogriješiti. Programeri moraju provjeravati da li se dogodio preliv, tj. gubitak podataka jer nijedna od tih funkcija na to ne upozorava. Ako je izvorni niz jednako dug ili dulji od odredišta, tj. traženog maksimalnog broja kopiranih znakova, `strncpy(3)` niz neće završiti s `null` pa to mora učiniti programer. Ako je izvorni niz kraći od odredišta, `strncpy(3)` će ostatak međuspremnika popuniti `null` znakovima što usporava rad sustava. Ako se argument traženog broja kopiranih znakova (`size_t n`) koristi u svrhu kopiranja dijela izvornog niza, onda niz

također neće biti završen s `null`, a u tom slučaju funkcija ne nudi nikakvu zaštitu protiv preliva međuspremnika.

Još jedno moguće rješenje je korištenje funkcije `sprintf(3)` koja uz predan format `%.*s` i maksimalnu duljinu izlaza omogućava upis niza u međuspremnik sa zaštitom od preliva.

```
#include <stdio.h>
int sprintf(char *str, const char *format, ...);
```

Ona završava niz s `null`, ali `null` znak nije uključen u maksimalnu duljinu izlaza pa na to treba paziti. Također je lako pogriješiti ako se koristi više parametara jer onda treba uzeti u obzir maksimalnu duljinu svih parametara zajedno.

```
char buf[MAXBUF];
// maksimalno zapisan MAXBUF - 1 znak + '\0' tj. null znak
sprintf(buf, "%.*s", MAXBUF - 1, "Ulazni niz znakova!");
```

Funkcija `snprintf(3)` je kroz normu ISO C99 uvedena kao alternativa za `sprintf(3)` u svrhu smanjivanja rizika od preliva međuspremnika.

```
#include <stdio.h>
int snprintf(char *str, size_t size, const char *format, ...);
```

Ta se funkcija ponaša poput `sprintf(3)` osim što kao parametar prima i maksimalan broj upisanih okteta te u slučaju skraćivanja podataka vraća broj okteta koji bi bili zapisani da je bilo dovoljno prostora. To omogućava jednostavnu detekciju skraćivanja podataka i dobru zaštitu od preliva međuspremnika. Nažalost su brojne implementacije ove funkcije odstupale od specifikacije pa je pisanje sigurnog i prenosivog izvornog teksta otežano.

```
char buf[MAXBUF];
// maksimalno zapisano MAXBUF okteta uključujući null
if (snprintf(buf, MAXBUF, "%s", "Ulazni niz znakova!") >= MAXBUF) {
        ... // dogodilo se skraćivanje ulaznih podataka (truncation)
}
... // podaci su stali u međuspremnik
```

Prilikom formatiranja ulaza funkcijama obitelji `scanf(3)` treba koristiti polje `width` koje omogućava postavljanje maksimalnog broja učitanih znakova. Također treba uzeti u obzir prostor za završni znak `null` [1][2][17][23].

```
char buf[MAXBUF];
char format[MAXFORMAT];
// nije moguće direktno koristiti definiranu konstantu MAXBUF
// stvoren niz znakova format "%(MAXBUF - 1)s", postavljen width
snprintf(format, MAXFORMAT, "%%%ds", MAXBUF - 1);
scanf(format, buf);
```

6.2.3. OpenBSD-ove strlcpy i strlcat

Funkcije `strlcpy(3)` i `strlcat(3)` nisu dio standardne C biblioteke, ali su dostupne u bibliotekama raznih UNIX okruženja uključujući BSD, Mac OS X, Solaris, Android i IRIX. Linuxov *glibc* ih nažalost ne podržava. Ove su funkcije namijenjene kao konzistentnija i sigurnija zamjena standardnih `strncpy(3)` i `strncat(3)` te također rade sa statički dodijeljenim međuspremnicima.

```
#include <string.h>
size_t strlcpy(char *destination, const char *source, size_t size);
size_t strlcat(char *destination, const char *source, size_t size);
```

Obje funkcije kao argument primaju punu veličinu međuspremnika (ne ostatak slobodnog prostora) koju neće prijeći te garantiraju završetak niza znakom `null`. Funkcija `strlcpy(3)` će kopirati najviše `(size - 1)`, a `strlcat(3)` najviše `(size - strlen(destination) - 1)` znakova, ostavljajući mjesta za znak `null`. Zbog takvog ponašanja, ove funkcije rade samo s nizovima koji završavaju s `null` znakom i pritom oslobađaju programera od računanja preostalog prostora međuspremnika nakon svake operacije. Za razliku od `strncpy(3)`, funkcija `strlcpy(3)` ne gubi vrijeme puneći ostatak odredišnog međuspremnika `null` znakovima. Također, `strlcpy(3)` i `strlcat(3)` kao povratnu vrijednost vraćaju ukupnu duljinu niza znakova koji su pokušale stvoriti što je korisno za detekciju skraćivanja izvornog niza [1][23].

```
char buf[MAXBUF];
// funkcije uvijek primaju ukupnu veličinu međuspremnika
// vraćaju duljinu niza koji bi bio stvoren da je bilo mjesta
if (strlcpy(buf, "Ulazni niz znakova!", MAXBUF) >= MAXBUF) {
        ... // dogodilo se skraćivanje ulaznih podataka (truncation)
}
if (strlcat(buf, "\nDodatan niz!", MAXBUF) >= MAXBUF) {
        ... // dogodilo se skraćivanje ulaznih podataka (truncation)
}
```

6.2.4. C++ std::string

Korištenjem standardnog razreda `std:string`, dodijeljen međuspremnik automatski raste po potrebi (dinamičko dodjeljivanje). Programski jezik izravno podržava naveden razred što omogućava laganu i prenosivu primjenu. Razred općenito sprječava preliv međuspremnika, no sadrži funkcije `c_str()` i `data()` koje vraćaju niz znakova poput onog standardnog u C-u. Ako se te funkcije nepažljivo koriste, vraća se problem preliva međuspremnika [1][23].

6.2.5. Ostale biblioteke

Nepravilno korištenje polja i nizova znakova je najčešći uzrok ranjivosti na preliv međuspremnika. Korištenje temeljito testiranih biblioteka za automatsko upravljanje međuspremnicima može značajno smanjiti utjecaj tog problema. Među sigurnijim i bržim bibliotekama za C/C++ su *SafeStr*, „*The Better String Library*“, *Vstr* i *Erwin*.

7. Oprezno oslanjanje na vanjske resurse

Gotovo svi programi u svom radu koriste neka druga računalna sredstva, dok rijetko koji program radi sam za sebe. Programi se radi pristupa resursima često oslanjaju na operacijski sustav, izvorne biblioteke ili neki drugi program. Detalji izvođenja poziva vanjskih resursa nisu sasvim očiti jer uključuju puno skrivenog rada u pozadini (npr. dinamičke biblioteke (engl. *dynamic library*)). Programer treba biti svjestan sredstava na koje se program oslanja i načina kojim se tim sredstvima pristupa [1][17].

7.1. Pozivanje sigurnih funkcija biblioteka

Prilikom pozivanja funkcija neke biblioteke visoke razine apstrakcije, nekad nije moguće utvrditi jesu li implementacije na svim sustavima jednake i koliko su sigurne. Specifikacija će takvu informaciju često izostaviti. Čak i ako se je moguće uvjeriti u sigurnost pojedine implementacije iz toga ne slijedi da su implementacije sučelja na drugim sustavima također sigurne. Javlja se konflikt između sigurnosnih principa te razvojnih principa apstrakcije i ponovnog korištenja. Pri kompilaciji ili instalaciji je moguće provesti ispitivanje sigurnosti implementacije neke funkcije pomoću alata poput *GNU Autoconf*. Na taj je način moguće zaustaviti instalaciju ako lokalno nisu implementirane sigurne verzije funkcija.

Ako se nije moguće uvjeriti u sigurnost funkcija biblioteke, treba samostalno implementirati vlastite verzije funkcija. Pri ponovnoj implementaciji neke funkcije, dobro je koristiti postojeća sučelja kako bi bilo moguće prebacivanje na implementaciju iz biblioteke gdje je ona sigurna [1][17].

7.2. Pozivanje s valjanim parametrima

Kad program poziva drugi program, treba provjeriti da su predani parametri valjane i očekivane vrijednosti. Mnogi su pozivi funkcija biblioteka implementirani na neočekivane načine te se mogu pozivati na ljusku. To znači da predavanje specijalnih znakova može biti problematično. Specijalni znakovi se ne interpretiraju kao podaci, već kao naredbe ili graničnici (npr. ljuska naredbenog retka ili interpreter SQL-a). Takvo se ponašanje može iskoristiti za ugrožavanje sustava.

Specijalni znakovi ljuske mogu utjecati na mnoge bitne sistemske pozive poput `popen(3)`, `system(3)`, `execlp(3)` i `execvp(3)` jer se pri izvođenju pozivaju na ljusku. Posebno opasan može biti `system(3)` jer koristi ljusku za *globbing*. Za stvaranje novog procesa se umjesto tih funkcija preporučuje direktno korištenje `execve(3)`.

WWW Security FAQ kao specijalne znakove ljuske navodi:

```
& ; ` ' \ " | * ? ~ < > ^ ( ) [ ] { } $ \n \r
```

U moguće problematične znakove se često uvrštavaju i: `!`, `#`, `-`, `\t` (tab), `' '` (razmak), `\v` (vertikalan razmak (engl. *vertical space*)), `\f` (*form feed*), te ostali upravljački znakovi poput `null`.

Kad se specijalni znakovi na sličan način koriste u ulazu SQL-a, to se često naziva *SQL injection*. Najbolje je definirati uski i siguran raspon mogućih znakova (npr. samo slovnobrojčani (engl. *alphanumeric*) znakovi). Ako je u ulazu nužno dopuštati specijalne znakove SQL-a, treba ih prije obrade što prije kodirati (npr. postotnim kodiranjem). Također je preporučljivo dodati navodnike na početak i kraj svakog ulaza kako bi se smanjila opasnost praznina (engl. *whitespace*) i raznih drugih podataka [1][2][17][24].

7.3. Pozivanje sučelja namijenjenih programerima

Iako program obično može pozvati bilo koji drugi program, često nije poželjno zvati programe namijenjene lakoj i intuitivnoj interakciji s ljudima (npr. uređivači teksta, programi za e-mail). Takvi programi obično imaju brojne opskurne ulaze te ih je teško u potpunosti nadzirati. Također često podržavaju prekidne kodove što napadaču može omogućiti neovlašten pristup sustavu. Umjesto programa namijenjenih korisnicima treba koristiti sučelja namijenjena programerima (engl. *application programming interfaces*, skraćeno APIs). Često postoje parametri programa ili druga sučelja koja će pružiti sigurniji pristup traženoj funkcionalnosti.

Povratne vrijednosti sistemskih poziva ne smiju biti ignorirane, već ih treba provjeriti i na njih odgovarajuće reagirati. Ako greška ne može biti mirno (engl. *gracefully*) razriješena, program treba sigurno pasti (engl. *fail safe*). Jedan od izvora takvih grešaka su ograničena sredstva sustava o kojima ovise gotovo svi sistemski pozivi, a korisnici na ta sredstva često mogu utjecati. Npr. veliki broj konkurentnih zahtjeva može iscrpiti resurse Web-

poslužitelja. Na *setuid/setgid* programe se ograničenja resursa mogu postaviti pomoću poziva `setrlimit(3)` ili `nice(2)` [1][2][17].

7.4. Skrivanje osjetljivih informacija

Osjetljive informacije na ulazu, izlazu ili u obliku pohranjenih podataka trebaju biti čitljive samo onome kome su namijenjene. One uključuju imena, e-mail adrese, kućne adrese, financijske izvještaje, brojeve kreditnih kartica, lozinke i ostale privatne informacije.

Baze podataka koje sadrže osjetljive informacije bi trebale biti enkriptirane. Enkripcija pruža razinu obrane kad je napadač već uspio doći do podataka, ali ne i do sigurnosnih ključeva. Upravo su u travnju 2011. godine u napadu na PlayStation Network i Qriocity poslužitelje dohvaćeni osobni podaci otprilike 77 milijuna korisničkih računa pohranjenih bez enkripcije. Obrana enkripcijom podataka nije dovoljna jer često sam poslužitelj može biti ugrožen.

Web-aplikacije bi također trebale enkriptirati svu osjetljivu komunikaciju s korisnicima. Obično se to radi korištenjem protokola HTTPS (HTTP preko SSL/TSL-a). Neki poslužitelji i aplikacije bilježe zahtjeve URI-a koji onda mogu biti vidljivi i drugima. Stoga za prijenos osjetljivih podataka nije pogodna HTTP metoda GET, već treba koristiti metodu POST [1][25][26].

8. Sigurno slanje povratnih informacija

Sigurno slanje povratnih informacija se uglavnom svodi na slanje samo neophodnih povratnih informacija. Nepouzdanom korisniku treba pružiti što manje informacija, posebno pri prijavi te neuspjehu ili padu programa. Detaljan izvještaj o događaju treba upisati u zapisnik, ali ne i na korisnikov zaslon. Sigurnosni sustav nikad ne bi smio ovisiti o neznanju napadača, no u praksi to služi kao dodatan sloj obrane dok se zakrpa ne stigne napisati i distribuirati [1].

8.1. Uključivanje komentara

U povratne informacije nije poželjno uključivati komentare ili dokumentaciju koje korisnici ne moraju vidjeti. To je većinom problem Web-aplikacija koje generiraju sadržaj (npr. HTML ili XML) gdje se komentari, namjerno ili ne, znaju uključiti u izlaz poslan korisniku. Takvi komentari mogu napadaču pomoći boljim uvidom u rad sustava, a također stvaraju dodatan promet [1].

8.2. Upravljanje izlaznim kanalom koji ne reagira

Korisnik može usporiti ili zaustaviti tok izlaznog kanala sigurnog programa. Npr. Web-preglednik se usred komunikacije može namjerno zaustaviti ili sporo odgovarati. Takve slučajeve bi siguran program trebao moći mirno rješavati. Poželjno je prije odgovaranja brzo otključati brave (engl. *release locks*) te postaviti vremensko ograničenje (engl. *time-out*) na rad mrežnih funkcija pisanja. U suprotnom sustav može biti ranjiv na napad uskraćivanjem usluga [1][2].

8.3. Upravljanje formatiranjem nizova znakova

Funkcije raznih programskih jezika primaju parametre koji upravljaju formatom izlaznog niza znakova. U programskom jeziku C to su funkcije poput `printf(3)`, `syslog(3)`, `err(3)` i `warn(3)`. Slično se u Pythonu postiže operacijom '`%`', a u Perlu je oponašano ponašanje funkcije `printf(3)` jezika C. U mnogim su programima i bibliotekama

definirane razne funkcije za formatiranje izlaza. One se često pozivaju na postojeće funkcije pa nasljeđuju njihove mane.

Ulazni niz znakova nepouzdanog korisnika ne smije biti interpretiran kao parametar formata. U suprotnom je program ranjiv na ulazni niz znakova koji predstavlja format. (engl. *uncontrolled format string, format string vulnerability*).

```
printf(niz_znakova_nepouzdanog_korisnika);       // pogrešno
printf("%s", niz_znakova_nepouzdanog_korisnika); // ispravno
```

Napadač koji upravlja formatom može izazvati razne probleme. Taj je problem najizraženiji u programskom jeziku C jer napadač može predugim nizom znakova formata izazvati preliv međuspremnika i dobiti potpun pristup sustavu. Također je moguće ubacivanje *conversion specifiera* (npr. `%s`, `%x`) bez odgovarajućih parametara što uzrokuje ispisivanje neočekivanih vrijednosti, odnosno podataka sa stoga ili iz ostale memorije. Posebno je opasan *conversion specifier* `%n` kojim se u pokazivač predan kao dodatan parametar funkcije upisuje broj ispisanih znakova. To napadaču omogućava pisanje u gotovo proizvoljnu lokaciju memorije [1][27].

9. Sigurnosni problemi specifični pojedinim programskim jezicima

U ovom poglavlju se razmatraju problemi specifični programskim jezicima C/C++, Java i skriptnim jezicima ljuske (engl. *shell scripting languages*) poput derivata *sh* i *csh*. Razni jezično specifični sigurnosni problemi se mogu riješiti primjenom sljedećih smjernica:

- Uključiti sve dostupne i značajne mehanizme upozorenja i zaštite gdje je to praktično. To se odnosi na mehanizme pri kompiliranju i pokretanju programa. Sigurni programi bi se trebali kompilirati bez ikakvih upozorenja.
- Koristiti siguran način rada (engl. *safe mode*) ako se nudi. Takav način rada podržavaju mnogi interpretirani jezici. Služi kao dodatan sloj zaštite (*defense in depth*), ali ne smije se smatrati apsolutnom obranom.
- Primjenjivati standardne konvencije jezika.
- Izbjegavati korištenje zastarjelih (engl. *deprecated*) funkcionalnosti jezika.
- Izbjegavati korištenje funkcionalnosti jezika koje je teško ispravno koristiti. Tu spadaju i „magične“ funkcije koje same pokušavaju zaključiti što treba napraviti jer napadač nekad može iskoristiti takvo ponašanje.
- Osigurati dostupnost i sigurnost infrastrukture jezika kao npr. biblioteke za vrijeme izvođenja (engl. *runtime library*).
- U jezicima s automatskim skupljanjem smeća, odnosno nereferenciranih podataka, (engl. *garbage collection*) se treba pobrinuti da osjetljive informacije poput lozinki i ključeva budu što prije pobrisane iz memorije.
- Treba točno poznavati semantiku korištenih operacija i provjeriti sve bitne povratne vrijednosti. Također je bitno paziti na značenje koje jezik pridaje vrijednostima s predznakom i bez predznaka. Semantiku operacija treba proučiti iz dokumentacije programskog jezika ili biblioteka [1].

9.1. C/C++

O sveprisutnim programskim jezicima C i C++ je već puno rečeno. U njima su napisani mnogi programi, uče se svugdje i brojni programeri će ih odabrati za vlastita rješenja. Iako temeljni dizajn oba jezika čini pisanje sigurnog izvornog teksta gotovo nemogućim, to se često traži. O upravljanju memorijom mora se brinuti programer, tipovi podataka nisu strogo provjeravani te je ranjivost na preliv međuspremnika teško u potpunosti izbjeći. Također, ne postoji ugrađena podrška za upravljanje iznimkama pa su kritične situacije često olako ignorirane. C i C++ su savršeni za programiranje sistemske podrške poput jezgre operacijskog sustava, no za pisanje sigurnih namjenskih programa je ipak preporučljivo odabrati programski jezik više razine.

Upravljanje memorijom se obavlja mehanizmima poput `malloc()`, `free()`, `new` i `delete` te je u potpunosti u rukama programera. Pogreške pri upravljanju memorijom su česta pojava i mogu stvoriti sigurnosne ranjivosti. Ako se korištena memorija dosljedno ne oslobađa, program zbog nedostatka slobodne memorije može prestati raditi. Takvo ponašanje sustav čini ranjivim na napade uskraćivanjem usluge. Mnogi sustavi nemaju ugrađenu zaštitu od pokušaja oslobađanja već slobodne memorije. Zbog toga dvostruko oslobađanje memorije (engl. *double free*) može prouzročiti rušenje programa ili stvoriti ranjivost. Verzije biblioteke *libc* novije od 5.4.23 te verzije *glibc* 2.x uključuju implementaciju funkcije `malloc()` čije se ponašanje može podešavati varijablom okruženja `MALLOC_CHECK_`. Kad je varijabla `MALLOC_CHECK_` postavljena, koristi se manje učinkovita implementacija koja može tolerirati i javiti jednostavne pogreške poput dvostrukog oslobađanja ili preliva za jedan oktet. Takve se greške ne moraju odmah manifestirati, već se mogu javiti kasnije u toku izvođenja programa. Ako se takve greške akumuliraju, pravi izvor problema se teško pronalazi pa je to tijekom razvoja programa korisna mogućnost.

U jezicima C i C++ se tipovi varijabli gotovo i ne provjeravaju pa je kod deklaracija tipova dobro biti što stroži. Kad vrijednost ne može biti negativna ili ne predstavlja broj, poželjno je koristiti varijable bez predznaka. Taj se problem često javlja kod korištenja varijabli tipa `char` koje predstavljaju broj s predznakom ako eksplicitno nije deklariran tip `unsigned char`. Kad se varijabla tipa `char` negativne vrijednosti pohrani u varijablu tipa `integer`, rezultat je negativan broj. Ako to ponašanje nije uzeto u obzir kod provjere vrijednosti,

može biti iskorišteno. Također bi trebalo koristiti `enum` funkcionalnost umjesto definiranja specijalnih cjelobrojnih vrijednosti (npr. `int`, `char`).

Mnoge se pogreške mogu izbjeći uključivanjem što je više moguće upozorenja kompajlera. Pri korištenju kompajlera *GCC* dobro je postaviti bar sljedeće zastavice:

```
gcc –O2 –pedantic –Wall –Wextra –Wstrict-prototypes –Wformat-
security
```

GCC je implementirao Stack-Smashing Protector (skraćeno SSP), ugrađenu zaštitu protiv razbijanja stoga. Zaštita svih funkcija izvornog teksta se uključuje zastavicom `-fstack-protect-all` i služi kao dodatan sloj obrane [1][17][28].

9.2. Java

Jedan od glavnih problema platforme Java je održavanje sigurnog okruženja za izvođenje mobilnih programa. Sigurnosna arhitektura Jave pruža zaštitu od zlonamjernih programa, ali ne može zaštititi od kvarova unutar pouzdanog izvornog teksta programa. Java jezik i virtualni stroj (engl. *virtual machine*) strogo provjeravaju tipove varijabli (engl. *type-safe*) te pružaju automatsko upravljanje memorijom i provjeravanje granica polja. Ti mehanizmi onemogućuju probleme poput napada prelivom međuspremnika i razbijanja stoga. Osnovni principi poput minimiziranja ovlasti i provjeravanja svih ulaza naravno vrijede, ali Java ima neke potencijalno skrivene ulaze o kojima također treba voditi brigu.

Tajne poput ključeva, algoritama i lozinki ne smiju biti skrivene unutar izvornog teksta ili podataka programa jer je do tih podataka modificiranim virtualnim strojem lako doći. Odlučnog napadača neće spriječiti ni *obfuscated* izvorni tekst programa.

Umjesto korištenja javnih (engl. *public*) varijabli treba implementirati pristupne metode (*gettere i settere*) kako bi se pristup mogao ograničiti. Treba koristiti privatne (engl. *private*) metode, a ako je to nemoguće onda treba dokumentirati razlog i osigurati zaštitu javnih metoda od zlonamjernih ulaza. *Static* varijable su dostupne iz ostalih razreda pa ih treba izbjegavati ili koristiti kao neizmjenjiva (engl. *immutable*) *final* polja. Općenito je dobro sve razrede i metode proglasiti kao *final* radi sprječavanja nasljeđivanja na nepredviđene načine. Pritom dolazi do sukoba između principa proširivosti i sigurnosti programa.

Izmjenjive (engl. *mutable*) objekte nikad ne bi trebalo slati potencijalno nepouzdanom dijelu programa jer onda mogu biti proizvoljno mijenjani. Polja su izmjenjiva iako njihov sadržaj možda nije. Izmjenjive objekte pristigle iz nepouzdanih izvora je važno klonirati (engl. *clone*) prije korištenja. Kad bi se izmjenjivom objektu izravno pristupalo, korisnik bi ga mogao izmijeniti nakon sigurnosne provjere, a tijekom korištenja.

Razredi koji sadrže osjetljive informacije se ne smiju moći serijalizirati (engl. *unserializable*), deserijalizirati (engl. *undeserializable*) ni klonirati (engl. *unclonable*). Serijalizacija razreda dopušta napadaču potpun uvid u podatke objekta. Ako je serijalizacija nužna, potrebno je barem osjetljiva polja proglasiti kao *transient* kako ne bi bila uključena u serijalizaciji. Deserijalizacija napadaču može omogućiti stvaranje instance razreda s proizvoljnim vrijednostima, tj. može se ponašati kao javni konstruktor. Mogućnost kloniranja objekata napadaču može omogućiti stvaranje instance razreda bez pokretanja konstruktora. Ako je kloniranje razreda neophodno, barem treba postavljanjem *final* modifikatora osigurati kako se metoda `clone()` neće moći zlonamjerno redefinirati [1][29].

9.3. Skriptni jezici ljuske

Zbog velike vjerojatnosti sigurnosnih propusta, mnogi sustavi ignoriraju *setuid/setgid* zastavicu u slučaju programa napisanih u skriptnim jezicima ljuske (npr. sh, csh, bash). To uzrokuje probleme s prenosivošću *setuid/setgid* skripata ljuske, no postoje i ostali razlozi za izbjegavanje pisanja sigurnih programa u takvim jezicima. Osnovna svrha ljuske je interaktivna i automatizirana komunikacija s korisnikom. Zbog toga napadač može iskoristiti neke manje očite ulaze ljuske. Skriptni jezici ljuske pogodni su za pisanje nesigurnih programa, tj. programa koji ulaze primaju iz pouzdanih izvora ili koji se izvršavaju s ovlastima korisnika.

U mnogim verzijama UNIX-a postoji *race condition* problem u jezgri koji čini *setid* skripte inherentno nesigurnima. Između vremena otvaranja datoteke za otkrivanje korištenog interpretera i vremena kad interpreter (*setid*) otvara datoteku za čitanje postoji interval u kojem datoteka može biti promijenjena. Taj se kvar s vremenom eliminira iz sustava, no ukoliko je prisutan moguće ga je jednostavno zaobići stvaranjem minimalnog C programa omotača koji će pokrenuti skriptu.

Na izvođenje programa mogu utjecati ulazi poput naziva datoteke ili sadržaja direktorija izvršnog programa. Napadač ne smije moći stvoriti datoteku koja u nazivu sadrži specijalne znakove ljuske, praznine, znakove novog retka ili nazive koji počinju s crticom. Vrijednosti varijabli okruženja poput PATH, ENV, BASH_ENV i IFS također mogu imati značajan utjecaj na rad skripte ili čak izazvati izvođenje proizvoljnog programa prije nego se skripta krene izvršavati.

Ako je pisanje sigurnih skripata ljuske ipak nužno, treba primjenjivati sljedeće smjernice:

- skriptu treba postaviti u direktorij u kojem neće moći biti promijenjena ili pomaknuta;
- varijable okruženja treba filtrirati ili u cijelosti obrisati prije pozivanja skripte;
- varijable okruženja PATH i IFS treba što ranije postaviti na poznate vrijednosti;
- radni direktorij također treba što ranije promijeniti;
- podatke treba čitati samo iz direktorija kojima pristup imaju samo pouzdani korisnici;
- nazive datoteka koji se predaju putem naredbenog retka uvijek treba staviti pod navodnike;
- kod pozivanja drugih naredaba treba koristiti opciju '—' kako bi se zabranio daljnji unos opcija;
- općenito treba primjenjivati strogu politiku propuštanja naziva datoteka, a posebno treba paziti na specijalne znakove i znakove novog retka.

Ograničene ljuske (engl. *restricted shells*) pridonose obrani (*defense in depth*), no u sigurnost koju pružaju se ne treba pouzdati. Takve ljuske je teško podesiti kako bi radile na željen način, a čak i onda ih je moguće oboriti. Svrha ljuske je pokretanje ostalih programa, ali pokretanjem novog programa ljuska gubi nadzor te novi program može dopustiti prethodno nedopustive radnje. Ograničene ljuske obično omogućavaju pokretanje malog broja programa, ali neki od njih (npr. mnogi uređivači teksta) mogu korisniku dopustiti pokretanje drugih programa. Čak i ako takvi programi nisu dostupni, često je iz ograničene ljuske ipak moguće izaći kombinacijom dostupnih programa i mogućnosti ljuske [1][30].

10. Praktični rad

U okviru rada proveden je niz ispitivanja izvornih tekstova programa dostupnih na fakultetskom Webu. U prvom poglavlju se kao jedan od razloga pisanja nesigurnih programa navodi nedostatak pažnje posvećen sigurnosti u školama. Ispitivanjem nekih izvornih tekstova s fakultetskog Weba će se pokušati provjeriti da li se pisanje sigurnih izvornih tekstova uči kroz primjere, ako već ne i eksplicitno kroz nastavni program.

Naglasak nije na metodama iskorištavanja sigurnosnih propusta, već na izbjegavanju stvaranja ranjivosti primjenom dobrih principa pisanja sigurnog izvornog teksta. Analiza je učinjena isključivo pregledavanjem teksta „redak po redak“ i primjenom smjernica opisanih u ovom radu. Radi jasnoće su neke ilustracije izvornog teksta skraćene, a nebitni dijelovi izbačeni.

10.1.Konverter

Ovaj program je pisan u jeziku Java, a namjena mu je pretvaranje tekstualne u XML datoteku u svrhu prikazivanja sadržaja XML datoteke preko Weba. Pretpostavka sigurnosnog okruženja je nepouzdana ulazna datoteka, što se daje naslutiti iz raznih provjera ulaza koje se u javljaju u izvornom tekstu. Zbog nepouzdanog ulaza koji uvjetuje izlaz ostalim korisnicima, ovaj program treba biti siguran.

Glavni ulaz, tj. tekstualna datoteka ima aplikaciji specifičan oblik pohrane podataka gdje su podaci zapisani kao čisti tekst i odvajani graničnicima (| ; ,).

```
#broj_kartice|jmbag|ime_korisnika|prezime_korisnika|
konfiguracija_kartice|serija_kartica|vrijedi_do|usluge_na_kartici

8745884582322454342 | 0036435543 | Vlatko | Pokos | SXAA | 1 |
110930 | 1,korisnik,01,0000 ; 2,e-indeks,01,0000 ; 3,menza,01,0001 |
```

Ulazni podatci su u internoj memoriju programa pohranjeni u obliku *model* razreda koji predstavljaju buduće XML elemente. Pretvorba iz teksta u model te iz modela u XML se provodi korištenjem *factory* razreda. Tijekom stvaranja modela se provode provjere nad ulazom, a nakon toga se modeli pretvaraju u izlazni XML dokument.

```
<cardList>
```

```
  <card cardID="8745884582322454342">
    <JMBAG>0036435543</JMBAG>
    <firstName>Vlatko</firstName>
    <lastName>Pokos</lastName>
    <conf>SXAA</conf>
    <series>1</series>
    <validTo>110930</validTo>
    <services>
      <service id="1" version="1">
        <name>korisnik</name>
        <owner>0</owner>
      </service>
      <service id="2" version="1">
        <name>e-indeks</name>
        <owner>0</owner>
      </service>
      <service id="3" version="1">
        <name>menza</name>
        <owner>1</owner>
      </service>
    </services>
  </card>
...
</cardList>
```

Razred `model.factory.Service` sadrži provjere ulaznih nizova znakova te iz njih stvara `model.Service` koji predstavlja service element XML dokumenta. Vlastitom metodom `isNumber(String)` se provjerava da ulazni nizovi znakova zaista predstavljaju broj.

```
private static boolean isNumber(String intS) {
 try {
       Long.parseLong(intS);
       return true;
 } catch(NumberFormatException e) {
       return false;
 }
}
```

Provjera zove metodu `Long.parseLong(String)` koja će baciti iznimku ako predan String ne predstavlja broj tipa `Long`. Problem nastaje jer se provjerava jedno, a očekuje drugo. Nakon provjere (`Long`) se taj ulaz smatra valjanim za interpretaciju kao tip `Byte` i `Integer`. To za veće brojeve nije istina pa se program ruši. Također, varijabla `model.Service.name` izravno predstavlja izlazno XML polje, a metoda `model.Service.setName(String)` se poziva bez ikakve provjere ulaza. Zbog toga je moguće slučajno ili namjerno ubacivanje štetnog sadržaja koji će dalje biti proslijeđen korisnicima.

```
package model.factory;
...
public class Service {
 public static model.Service fromText(String input)  {
       model.Service usluga = new model.Service();
       StringTokenizer tok = new StringTokenizer(input, ",");
```

```
        //šifra usluge
        String sifraS = tok.nextToken().trim();
        if( isNumber(sifraS) == false )
              return null;
        usluga.setServiceID(Byte.parseByte(sifraS));

        //ime usluge
        usluga.setName(tok.nextToken().trim());
        ...
        //vlasnik usluge
        String ownerS = tok.nextToken().trim();
        if( isNumber(ownerS) == false )
              return null;
        usluga.setOwner(Integer.parseInt(ownerS));

        return usluga;
  }
...
}
```

Razred `model.Service` u *setter* metodama nema nikakve provjere ni ne vraća povratne vrijednosti, već se oslanja na prethodno opisan razred `model.factory.Service` kako bi ga sigurno postavio.

```
package model;
public class Service {
  ...
  public void setServiceID(byte serviceID) {
        this.ID = serviceID;
  }
  public void setName(String name) {
        this.name = name;
  }
  public void setVersion(byte version) {
        this.version = version;
  }
  public void setOwner(int owner) {
        this.owner = owner;
  }
  ...
}
```

Razred `model.factory.Card`, za razliku od `model.factory.Service`, nema sigurnosne provjere, već se oslanja na *setter* metode razreda `model.Card`. Te metode u slučaju neuspjeha provjere vraćaju vrijednost `false`, no njihove povratne vrijednosti uopće nisu provjeravane.

```
package model.factory;
...
public class Card {
  public static model.Card fromText(String input) {
        model.Card kartica = new model.Card();
        //tokenizacija stringa
```

```
        StringTokenizer st = new StringTokenizer(input, "|");

        //broj kartice
        kartica.setCardId(st.nextToken().trim());
        //jmbag
        kartica.setJmbag(st.nextToken().trim());
        //ime
        kartica.setFirstName(st.nextToken().trim());
  ...
        return kartica;
  }
...
}
```

Razred `model.Card` u svojim pristupnim metodama (*setteri* i *getteri*) sadrži provjere ulaznih nizova koji služe postavljanju objekta. Provjere su nepotpune, tj. provjeravaju se samo duljine ulaznih nizova te se, ukoliko one odgovaraju očekivanim, ulaz podrazumijevano smatra valjanim. Ostala polja uopće nisu provjerena pa je stoga ulaz čitavog razreda, a time izravno i konačan izlaz, ranjiv na ubacivanje proizvoljnih štetnih nizova znakova.

```
package model;
public class Card {
...
 public boolean setCardId(String id) {
        //provjera duljine ID-a
        if( id.length() != 19 )
              return false;
        this.id = id;
        return true;
 }
 public boolean setJmbag(String jmbag) {
        //provjera duljine JMBAG-a
        if( jmbag.length() != 10 )
              return false;
        this.jmbag = jmbag;
        return true;
 }
 public boolean setConfiguration(String configuration) {
        //duljina mora biti 4 znaka
        if( configuration.length() != 4)
              return false;
        this.configuration = configuration;
        return true;
 }
 public boolean setFirstName(String firstName) {
        this.firstName = firstName;
        return true;
 }
 public boolean setLastName(String lastName) {
        this.lastName = lastName;
        return true;
 }
...
}
```

Uočljivo je da se razne provjere ne nalaze konzistentno na istoj razini programskog dizajna, već su razbacane po projektu, što otežava sigurnosne preglede poput ovog. Provjere ulaza za postavljanje objekta `model.Service` se nalaze u razredu `model.factory.Service`, dok se za sigurno postavljanje objekta `model.Card` brinu njegove vlastite *setter* metode.

Vidljivo je i da provjere nisu *deny-by-default*, tj. provjerava se za nedozvoljene unose, a ako te provjere uspješno prođu, ulaz se smatra valjanim. To nije dobra praksa jer je često teško pokriti sva moguća nedozvoljena stanja ulaza, što je upravo u ovom izvornom tekstu slučaj. Umjesto toga bi se trebalo provjeravati za ispravan unos, a podrazumijevano odbacivati ostalo.

Ulazi nisu provjeravani za specijalne znakove koje koristi ulazna datoteka. Zbog toga je još opasnije „slijepo" korištenje razreda `StringTokenizer` za rastavljanje ulaznih nizova po graničnicima. Zbog toga ubacivanje neočekivanih specijalnih znakova u ulaz može izazvati rušenje aplikacije.

Izvorni tekst sadrži neke provjere i upravljanje iznimkama, ali dizajn provjera nije dobar pa nisu propuštane samo valjane vrijednosti. Provjere su nekonzistentno raspoređene po razredima što otežava pregled izvornog teksta. Neki su dijelovi ulaza provjeravani, a neki nisu. Izvorni tekst nije siguran ni primjenjiv u stvarnosti.

10.2.Mojping

Ovaj program je pisan u jeziku C, a namjena mu je emulacija osnovne funkcionalnosti sveprisutnog *ping* alata, tj. upravljanje ICMP *echo*/*reply* porukama. Pretpostavka sigurnosnog okruženja je nepouzdan lokalni korisnik. Program se zbog pristupa „*raw socket*" sučelju pokreće kao *setuid/setgid* pa mora biti siguran. Glavni ulazi su parametar mete s naredbenog retka (`argv[1], host`) i povratni *echo reply* paket.

Globalna varijabla `pid` se postavlja na identifikator procesa (engl. *process identifier*, skraćeno PID) sistemskim pozivom `getpid()`. Taj poziv je uvijek uspješan pa ga ne treba provjeravati.

```
int main(int argc, char *argv[])
{
    struct addrinfo *ai; // pokazivač na informacije o meti
    char *host; // pokazivač na niz znakova koji predstavlja metu

    if (argc != 2) // provjera
```

```
        errx(1,"usage: %s <hostname>",argv[0]);
    host = argv[1]; // postavljanje mete

    pid = getpid();
    Signal(SIGALRM, sig_alrm);
    ai = Host(host);
    ...
```

Omotačem funkcije `signal()` se upravljanje signalom (engl. *signal handling*) SIGALRM postavlja na prekidnu funkciju `sig_alrm()`, provjeravajući pritom povratnu vrijednost i moguću grešku sistemskog poziva. Korištenje funkcije `signal()` se zbog različitog ponašanja u raznim UNIX okruženjima (manja prenosivost) obeshrabruje te se preporučuje korištenje funkcije `sigaction()`. Također je vjerojatno poželjno pozvati `err()` umjesto `warn()` kako bi program u slučaju greške završio, inače je nastavak programa besmislen.

```
void *Signal(int signo, void *func)
{
    void *sigfunc;
    if ((sigfunc = signal(signo, func)) == SIG_ERR)
        warn("signal error");
    return (sigfunc);
}
```

Funkcija `Host()` je omotač za `getaddrinfo()`. Ona dohvaća informacije o meti te sigurno zbrinjava povratnu vrijednost i moguću grešku. Funkcija `memset()` se na prenosiv način poziva s argumentom `sizeof(var)`. Ulazni niz znakova na koji pokazuje `host` je poslužio svrsi i ne bi više smio imati utjecaja na program.

```
struct addrinfo * Host(const char *host)
{
    int n;
    struct addrinfo hints, *res;

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_flags = AI_CANONNAME;
    hints.ai_family = AF_INET;
    hints.ai_socktype = 0;

    if ((n = getaddrinfo(host, NULL, &hints, &res)) != 0)
        err(1,"getaddrinfo(%s): %s",host, gai_strerror(n));

    return (res);  /* pokazivac na prvi element vezane liste */
}
```

Varijabla `ai` pokazuje na informacije o meti te se one korištenjem funkcije `printf()` uz *conversion specifier* `%s` sigurno (format nije varijabilan niz znakova) ispisuju na standardni izlaz.

```
int main(int argc, char *argv[])
```

```
{
    ...

    printf("PING %s (%s): %d data bytes\n", ai->ai_canonname,
              Inet_ntop(ai->ai_addr, ai->ai_addrlen), datalen);

    sasend = ai->ai_addr;
    sarecv = Calloc(1, ai->ai_addrlen);
    salen = ai->ai_addrlen;

    readloop();

    exit(0);
}
```

Za pretvaranje IP adrese u standardni zapis koristi se siguran omotač funkcije `inet_ntop()`. Vraća se pokazivač na *static* polje maksimalne moguće veličine traženog zapisa (`INET_ADDRSTRLEN`). U slučaju greške se sigurno vraća pokazivač `NULL` kako ne bi bio vraćen i ispisan nepoznat sadržaj memorije.

```
char *Inet_ntop(const struct sockaddr *sa, socklen_t salen)
{
    static char str[INET_ADDRSTRLEN];/* Unix domain is largest */
    struct sockaddr_in *sin = (struct sockaddr_in *)sa;
    if (inet_ntop(AF_INET, &sin->sin_addr, str, sizeof(str))==NULL)
{
       warn("inet_ntop error");      /* inet_ntop postavlja errno */
       return (NULL);
    }
    return (str);
}
```

Globalni pokazivači `sasend` i `salen` se postavljaju na dobivene informacije o meti. Globalni pokazivač `sarecv` pokazuje na memoriju dodijeljenu sigurnim omotačem funkcije `calloc()`.

```
void * Calloc(size_t n, size_t size)
{
    void *ptr;
    if ((ptr = calloc(n, size)) == NULL)
       warn("calloc error");
    return (ptr);
}
```

Na ovakvu grešku nema smisla upozoravati s `warn()`, već je umjesto toga vjerojatno bolje koristiti `err()` kako bi program nakon izvještaja završio.

Funkcija `readloop()` predstavlja glavnu funkcionalnost programa. Umjesto funkcije `exit()` je većinom preporučeno koristiti `return 0` za izlaz iz `main()` funkcije. Memorija

zauzeta s `Host()` i `Calloc()` ne treba biti oslobođena jer joj se globalnim pokazivačima pristupa tijekom cijelog izvršavanja programa.

```
void readloop(void)
{
    char          recvbuf[BUFSIZE];
    socklen_t     len;
    ssize_t       n;
    struct timeval  tval;

    sockfd = Socket(sasend->sa_family, SOCK_RAW, IPPROTO_ICMP);

    sig_alrm(SIGALRM);          /* salji prvi ICMP paket */

    ...
}
```

Statički se dodjeljuje memorija za dolazni spremnik `recvbuf` te se sigurnim omotačem funkcije `socket()` stvara spojna točka (engl. *socket*). U slučaju neuspjeha omotača `Socket()`, dogodit će se i neuspjeh sljedeće funkcije `recvfrom()` pa bi `warn()` zbog jasnoće trebao biti zamijenjen s `err()` kako bi program završio s pravim izvještajem greške.

```
int Socket(int family, int type, int protocol)
{
    int n;
    if ((n = socket(family, type, protocol)) < 0)
       warn("socket error");
    return (n);
}
```

Pozivom funkcije `sig_alrm(SIGALRM)` šalje se prvi ICMP *echo* paket te se aktivira alarm koji će prekinuti čekanje odgovora i izazvati sljedeće slanje. Na taj način alarm djeluje i kao vremensko ograničenje na funkciju čitanja odgovora `recvfrom()`.

```
void sig_alrm(int signo)
{
    send_icmp();          /* posalji jednu ICMP poruku */
    alarm(1);
    return;               /* i nastavi s radom ... */
}
```

Funkcija `send_icmp()` precizno postavlja polja ICMP paketa koristeći izlazni međuspremnik `sendbuff` kao strukturu `struct icmp` te zove omotač funkcije `sendto()` koji obrađuje pogreške. Trenutno vrijeme se u obliku strukture `struct timeval` upisuje u ICMP polje za podatke i računa se zbroj za provjeru (engl. *checksum*). Varijabla `datalen` je fiksna i sadrži veličinu polja za podatke (56).

```
void send_icmp(void)
{
    int          len;
    struct icmp    *icmp;

    icmp = (struct icmp *)sendbuf;
    icmp->icmp_type = ICMP_ECHO;
    icmp->icmp_code = 0;
    icmp->icmp_id = pid;
    icmp->icmp_seq = nsent++;
    Gettimeofday((struct timeval *)icmp->icmp_data);

    len = 8 + datalen;          /* kontrolni zbroj i podaci */
    icmp->icmp_cksum = 0;
    icmp->icmp_cksum = in_cksum((u_short *) icmp, len);

    Sendto(sockfd, sendbuf, len, 0, sasend, salen);
}

void Sendto(int fd, const void *ptr, size_t nbytes, int flags,
            const struct sockaddr *sa, socklen_t salen)
{
    if (sendto(fd, ptr, nbytes, flags, sa, salen) != nbytes)
       warn("sendto error");
}
```

U funkciji `readloop()` se na odgovor čeka funkcijom `recvfrom()`, a odgovor se obrađuje u funkciji `proc_icmp()`. Funkciji `recvfrom()` su predani sigurni argumenti veličine spremnika, a povratna vrijednost i pogreške se pravilno obrađuju. U slučaju prekida se nastavlja čekati odgovor, a novi ICMP *echo* paket je već poslan u prekidnoj funkciji.

```
void readloop(void)
{
  ...

    for (;;) {
       len = salen;
       n = recvfrom(sockfd,recvbuf,sizeof(recvbuf),0,sarecv,&len);
       if (n < 0) {
             if (errno == EINTR)
                   continue;
             else
                   err(1,"recvfrom error");
       }
       Gettimeofday(&tval);
       proc_icmp(recvbuf, n, &tval);
    }
}
```

Prilikom obrade odgovora u funkciji `proc_icmp()` se još provjeravaju slučajevi greške poput stizanja premalog paketa, paketa koji nisu ICMP *echo reply*, paketa koji nemaju valjan identifikator i paketa koji nisu vratili bar 16 okteta podataka (veličina `struct timeval` na 32bit arhitekturama). Bolje bi bilo koristiti provjeru sa `sizeof(struct`

`timeval)` jer veličina strukture varira ovisno o arhitekturi računala (sadrži dvije varijable tipa `long int` koji su veličine 8 okteta u 64bit UNIX okruženjima). Provjere pokušavaju detektirati moguće pogrešne ulaze, a inače nastavljaju s radom (*allow-by-default*), što je općenito loša praksa. Primljen paket se samo interpretira i sigurnim pozivom `printf()` se ispisuju zanimljiva polja. U ovom se programu zbog nedostatka strogosti ne može dogoditi ništa loše, osim primanja i ispisivanja nekoliko neočekivanih vrijednosti.

```
void proc_icmp(char *ptr, ssize_t len, struct timeval *tvrecv)
{
  ...
     if ((icmplen = len - hlen1) < 8)
        err(1,"icmplen (%d) < 8", icmplen);

     if (icmp->icmp_type == ICMP_ECHOREPLY) {
        if (icmp->icmp_id != pid)
              return;           /* nije odgovor na nass ECHO_REQUEST
*/
        if (icmplen < 16)
              err(1,"icmplen (%d) < 16", icmplen);

        tvsend = (struct timeval *)icmp->icmp_data;
        tv_sub(tvrecv, tvsend);
        rtt = tvrecv->tv_sec * 1000.0 + tvrecv->tv_usec / 1000.0;

        printf("%d bytes from %s: seq=%u, ttl=%d, rtt=%.3f ms\n",
               icmplen, Inet_ntop(sarecv, salen),
               icmp->icmp_seq, ip->ip_ttl, rtt);
     }
}
```

Izvorni tekst je pisan na siguran i minimalistički način. Granice međuspremnika su poštovane, a povratne vrijednosti sistemskih poziva obrađene. Neke su veličine tipova varijabli upisane izravno u izvorni tekst pa može biti problema s prenosivosti. Također, neke se ključne greške samo izvještavaju, ali se program tada ne gasi već pada tek na sljedećoj provjeri koja ga gasi. Zbog jednostavnosti kritičnog dijela izvornog teksta, konzistentne primjene sigurnih omotača i brzog rješavanja nepouzdanih ulaza za ovaj je program ipak moguće reći da vjerojatno nema sigurnosnih ranjivosti.

10.3.Mojdaytime

Ovaj je program pisan u jeziku C, a namjena mu je jednostavno oglašavanje trenutnog vremena preko mreže. Pretpostavka sigurnosnog okruženja je nepouzdan udaljen korisnik na mreži. Program mora biti siguran jer radi kao demon poslužitelj, a zahtjeve dobiva od nepouzdanih korisnika s mreže.

Dio izvornog teksta koji je bitan za sigurnost kreće s prihvaćanjem veze korisnika.

```
int main(int argc, char **argv)
{
 ...

 for (;;) {
       len = addrlen;
       connfd = Accept(listenfd, cliaddr, &len);
       warn("connection from %s", Sock_ntop(cliaddr, len));

       ticks = time(NULL);
       snprintf(buff, sizeof(buff), "Moj daytime
server:\n%.24s\r\n",
                   ctime(&ticks));
       Write(connfd, buff, strlen(buff));

       Close(connfd);
    }
}
```

Opisnik datoteke spojne točke se dohvaća sigurnim omotačem funkcije `accept()` koji se brine o mogućim greškama. Osim poništavanja veze i pogreške u protokolu, greške izazivaju gašenje programa uz izvještaj. Također, izvorni tekst nepotrebno sadrži notornu `goto` naredbu.

```
int Accept(int fd, struct sockaddr *sa, socklen_t * salenptr)
{
    int n;
again:
    if ((n = accept(fd, sa, salenptr)) < 0) {
       if (errno == EPROTO || errno == ECONNABORTED)
           goto again;
       else
           err_quit("accept error");
    }
    return (n);
}
```

Trenutno vrijeme se dohvaća funkcijom `time()` te se rezultat ispisuje u međuspremnik `buff` uz sigurne pozive `snprintf()` i `ctime()`. Moguće greške poziva `time()` i `ctime()` se ne provjeravaju. U slučaju greške se umjesto formatiranog vremena ispisuje niz znakova „null“. Novostvorena poruka se šalje sigurnim omotačem funkcije `write()` te se veza zatvara omotačem funkcije `close()`. Ti omotači gase poslužitelj pri svakoj greški.

```
void Write(int fd, void *ptr, size_t nbytes)
{
    if (write(fd, ptr, nbytes) != nbytes)
       err_quit("write error");
}

void Close(int fd)
```

```
{
    if (close(fd) == -1)
       err_quit("close error");
}
```

Ovaj jednostavan izvorni tekst poslužitelja nema puno mjesta za pogreške jer ulaz nepouzdanog korisnika nema gotovo nikakav utjecaj. Greške sistemskih poziva se rješavaju gašenjem programa, što možda i nije potrebno. Program je eventualno (ovisno o postavkama UNIX okruženja) ranjiv na napade uskraćivanjem usluga poput preplavljivanja paketima SYN (engl. *SYN flood*).

10.4.Nweb

Ovaj je program pisan u jeziku C, a služi kao jednostavan Web-poslužitelj. Pretpostavka sigurnosnog okruženja je nepouzdan udaljen korisnik na mreži. Program mora biti siguran jer radi kao demon poslužitelj, a zahtjeve prima od nepouzdanih korisnika s mreže. Glavni nepouzdan ulaz je zahtjev za Web-stranicom poslan nakon uspostavljanja veze.

Dio izvornog teksta koji utječe na sigurnost počinje s prihvaćanjem veze korisnika.

```
main(int argc, char **argv)
{
    int i, port, pid, listenfd, socketfd, hit;
    size_t length;
    char   *str;
    static struct sockaddr_in cli_addr;
    static struct sockaddr_in serv_addr;

    ...

    for (hit = 1;; hit++) {
       length = sizeof(cli_addr);
       if ((socketfd = accept(listenfd,(struct sockaddr *)
                   &cli_addr,&length)) < 0)
             mojlog(ERROR, "system call", "accept", 0);

       if ((pid = fork()) < 0) {
           mojlog(ERROR, "system call", "fork", 0);
       } else {
           if (pid == 0) {     /* child */
             (void)close(listenfd);
             web(socketfd, hit);     /* never returns */
           } else {            /* parent */
             (void)close(socketfd);
           }
       }
    }
}
```

Povratne vrijednosti poziva `accept()` i `fork()` se primjereno obrađuju te u slučaju greške program završava uz izvještaj. Povratna vrijednost funkcije `close()` se ne provjerava, što nije zabrinjavajuće. Konkurentni zahtjevi se obrađuju uz pomoć poziva `fork()`, a novi proces zove funkciju `web()` u kojoj poslužuje zahtjev. Funkcija `mojlog()` obrađuje greške te se ponaša ovisno o predanim parametrima.

```
void mojlog(int type, char *s1, char *s2, int num)
{
    int fd;
    char logbuffer[BUFSIZE * 2];

    switch (type) {
    case ERROR:
        (void)sprintf(logbuffer, "ERROR: %s:%s Errno=%d exiting
pid=%d", s1, s2, errno, getpid());
        break;
    case SORRY:
        (void)sprintf(logbuffer, "<HTML><BODY><H1>nweb Web Server
Sorry: %s %s</H1></BODY></HTML>\r\n", s1, s2);
        (void)write(num, logbuffer, strlen(logbuffer));
        (void)sprintf(logbuffer, "SORRY: %s:%s", s1, s2);
        break;
    case LOG:
        (void)sprintf(logbuffer, " INFO: %s:%s:%d", s1, s2, num);
        break;
    }
    /* no checks here, nothing can be done a failure anyway */
    if ((fd = open("nweb.log",O_CREAT|O_WRONLY|O_APPEND,0644)) >= 0)
{
        (void)write(fd, logbuffer, strlen(logbuffer));
        (void)write(fd, "\n", 1);
        (void)close(fd);
    }
    if (type == ERROR || type == SORRY)
        exit(3);
}
```

Statički dodijeljen međuspremnik `logbuffer` je dvostruko veći od ostalih međuspremnika (`2*BUFSIZE`). Time se osigurava dovoljan prostor za rad sa svim mogućim ulazima te se omogućava korištenje funkcije `sprintf()` za formatiranje nizova znakova u odredišne međuspremnike. Funkciju `sprintf()` je u ovom slučaju moguće koristiti bez provjera granica međuspremnika jer je međuspremnik dimenzioniran kako bi u njega mogao stati svaki ulaz. Radi bolje proširivosti (npr. obrađivanje novih ulaza) bi bilo dobro provjeravati da se granice međuspremnika `logbuffer` poštuju.

Funkcija `web()` predstavlja glavnu funkcionalnost poslužitelja.

```
/* this is a child web server process, so we can exit on errors */
void web(int fd, int hit)
{
    int j, file_fd, buflen, len;
```

```
    long i, ret;
    char *fstr;
    static char buffer[BUFSIZE + 1]; /* static so zero filled */

    ret = read(fd, buffer, BUFSIZE); /* read Web request in one go
*/
    if (ret == 0 || ret == -1) {/* read failure stop now */
       mojlog(SORRY, "failed to read browser request", "", fd);
    }
    if (ret > 0 && ret < BUFSIZE)     /* return code is valid chars
*/
       buffer[ret] = 0;  /* terminate the buffer */
    else
       buffer[0] = 0;
 ...
}
```

Međuspremnik `buffer` veličine `BUFSIZE + 1` se sigurno inicijalizira popunjen nulama. S takvim se međuspremnikom može koristiti kao da je veličine `BUFSIZE`, a zadnji oktet će garantirati završetak niza `null` znakom. Na taj način ispis sadržaja međuspremnika ne može uzrokovati ispis memorije izvan granica (do prvog znaka `null`). Preliv međuspremnika `logbuffer` u funkciji `mojlog()` je onemogućen ograničavanjem međuspremnika `buffer` na veličinu `BUFSIZE`. Provodi se dodatna provjera da je cijeli ulaz stao u međuspremnik te se niz završava znakom `null`.

```
void web(int fd, int hit)
{
...
    for (i = 0; i < ret; i++)   /* remove CF and LF characters */
       if (buffer[i] == '\r' || buffer[i] == '\n')
             buffer[i] = '*';
    mojlog(LOG, "request", buffer, hit);

    if (strncmp(buffer, "GET ", 4) && strncmp(buffer, "get ", 4))
       mojlog(SORRY,"Only simple GET operation
supported",buffer,fd);

    for (i = 4; i < BUFSIZE; i++) {
       if (buffer[i] == ' ') {
           buffer[i] = 0;
        break;
       }
    }

    /* check for illegal parent directory use .. */
    for (j = 0; j < i - 1; j++)
       if (buffer[j] == '.' && buffer[j + 1] == '.')
             mojlog(SORRY, "Parent directory (..) path names not
supported", buffer, fd);

...
}
```

Slijedi zamjena znakova novog retka (`\r`, `\n`) sa zvjezdicama (`*`) kako se zapis o zahtjevu ne bi lomio preko redova. Provjerava se da ulaz sadrži jedinu podržanu HTTP metodu GET, a ostali se ulazi sigurno odbacuju. Funkcija `mojlog()` izvještava o greškama te prekida izvođenje programa. Niz se završava znakom `null` na mjestu drugog razmaka, tj. nakon „GET URL“ dijela zahtjeva kako bi se olakšala daljnja obrada. Upit se provjerava za nelegalan zahtjev koji sadrži '..' te se u slučaju takvog ulaza šalje odgovor s opisom greške i prekida izvođenje.

```
void web(int fd, int hit)
{
...
    /* work out the file type and check we support it */
    buflen = strlen(buffer);
    fstr = (char *)0;
    for (i = 0; extensions[i].ext != 0; i++) {
       len = strlen(extensions[i].ext);
       if (!strncmp(&buffer[buflen - len], extensions[i].ext, len))
{
             fstr = extensions[i].filetype;
             break;
       }
    }

    if (fstr == 0)
       mojlog(SORRY, "file extension type not supported", buffer,
fd);

    if ((file_fd = open(&buffer[5],O_RDONLY)) == -1)
       mojlog(SORRY, "failed to open file", &buffer[5], fd);

    mojlog(LOG, "SEND", &buffer[5], hit);

    (void)sprintf(buffer, "HTTP/1.0 200 OK\r\nContent-Type:
%s\r\n\r\n",fstr);
    (void)write(fd, buffer, strlen(buffer));

    /* send file in 8KB block - last block may be smaller */
    while ((ret = read(file_fd, buffer, BUFSIZE)) > 0) {
       (void)write(fd, buffer, ret);
    }
 #ifdef LINUX
    sleep(1);            /* to allow socket to drain */
 #endif
    exit(1);
}
```

Završetak naziva zahtijevane datoteke se uspoređuje s poznatim i dopuštenim nastavcima. Ako tražena datoteka nije podržanog nastavka, zahtjev se sigurno odbija i šalje se odgovor s opisom greške. Tražena datoteka se otvara funkcijom `open()` uz provjeru povratne vrijednosti i obradu grešaka. Funkcijom `sprintf()` se u međuspremnik `buffer` kopira HTTP zaglavlje koje će u njega sigurno stati. Zaglavlje se šalje funkcijom `write()` te se u

međuspremnik `buffer` funkcijom `read()` počne čitati iz tražene datoteke. Čita se uz sigurno ograničenje maksimalne primljene veličine `BUFSIZE`, a pročitano se korisniku šalje funkcijom `write()`. Povratne vrijednosti funkcije `write()` se pri slanju HTTP zaglavlja i datoteke ne provjeravaju, već se moguće greške ignoriraju.

Ovaj izvorni tekst je pisan kako bi bio siguran i jednostavan. Restriktivna politika prihvaćanja metoda i datoteka omogućava lagano odbijanje nevaljanih ulaza. Obrada grešaka se nalazi na jednom mjestu (funkcija `mojlog()`), što je pozitivno. Nije u potpunosti provjeren ulaz već su uzete u obzir samo neke moguće greške (*allow-by-default*), što je česta, ali loša praksa. Zbog toga prolaze nevaljani HTTP zahtjevi kojima nedostaju određeni dijelovi zaglavlja ('`/`' ili '`HTTP/1.X`'). Tako prolazi zahtjev poput npr. „GET ABCzip“ (nedostaje početni znak '`/`'), a interpretira se kao dohvat datoteke „BCzip“.

Zaključak

Razmatrana tematika pokriva opsežno područje pa je s obzirom na ciljan volumen rada bilo teško ujednačiti razinu apstrakcije i odabrati dijelove koji bi zadovoljavajuće predstavili problem. Ovaj rad zato nije dovoljan za stvaranje sigurnih programa, već pokušava sigurnosno osvijestiti programera i dati mu općenit uvid u kompleksnost i težinu problema. Programer mora garantirati sigurnost sustava za sve ulaze u svim okolnostima, dok je napadaču često dovoljno pronaći jedan kvar da čitav sustav ugrozi. Inženjer sigurnih programa zato razmišlja poput napadača, stalno prati sigurnosnu scenu, čita i primjenjuje znanja iz literature, koristi sistematske pristupe uvjeravanja u sigurnost i intimno poznaje sredstva s kojima raspolaže.

U moru zadataka i primjera s podrazumijevanim okruženjima, bajnim algoritmima i novim tehnologijama, provjere ulaza i povratnih vrijednosti su često prve zanemarene. Uz nesigurne primjere izvornih tekstova s besmisleno polovičnim provjerama i sakrivenim detaljima se lako uljuljati u lažan osjećaj sigurnosti, što je možda gore nego da provjera uopće nema. Ipak, ispitivanja izvornih tekstova programa pokazuju da je zbilja moguće imati stvarne primjere sigurnog izvornog teksta kao dio školskog programa i da oni postoje. Ohrabrujuće je da se studijem barem kroz primjere ostalih predmeta ipak upoznaje s problemima pisanja sigurnog izvornog teksta, kad već nedostaje predmeta posvećenih tematici.

Literatura

[1] WHEELER, D. A.: Secure Programming for Linux and Unix HOWTO, s Interneta, http://www.dwheeler.com/secure-programs/, 17. svibnja 2011.

[2] GALVIN, P.: The UNIX Secure Programming FAQ, s Interneta, http://sunsite.uakom.sk/sunworldonline/swol-08-1998/swol-08-security.html, 17. svibnja 2011.

[3] LEVY, E. (ALEPH ONE): Learning Security [SUMMARY], Bugtraq, 17. prosinca 1998., s Interneta, http://seclists.org/bugtraq/1998/Dec/60, 17. svibnja 2011.

[4] RIJMEN, V.: LinuxSecurity.com speaks with AES winner, s Interneta, http://www.linuxsecurity.com/content/view/117552/49/, 17. svibnja 2011.

[5] LEVY, E. (ALEPH ONE): Wide Open Source, s Interneta, http://www.securityfocus.com/news/19, 17. svibnja 2011.

[6] DIFFIE, W.: Risky business: Keeping security a secret, s Interneta, http://www.zdnet.com/news/risky-business-keeping-security-a-secret/127072, 17. svibnja 2011.

[7] VIEGA, J.: The Myth of Open Source Security, s Interneta, http://www.developer.com/tech/article.php/626641/The-Myth-of-Open-Source-Security.htm, 17. svibnja 2011.

[8] CALOYANNIDES, M.; WITTEN, B.; LANDWEHR, C.: Does Open Source Improve System Security, *IEEE Software*, pp. 57-61, rujan 2001.

[9] Common Criteria for Information Technology Security Evaluation (CC) v3.1, ISO/IEC 15408, s Interneta, http://www.commoncriteriaportal.org/cc/, 17. svibnja 2011.

[10] SALTZER, J. H.; SCHROEDER, M. D.: The Protection of Information in Computer Systems, s Interneta, http://www.multicians.org/ProtInf/, 18. svibnja 2011.

[11] NEUMANN, P. G.: Principled Assuredly Trustworthy Composable Architectures, s Interneta, http://www.csl.sri.com/users/neumann/chats4.html, 18. svibnja 2011.

[12] CASSAT, P. C.; SALOMON, K. D.: Gramm-Leach-Bliley Act Safeguards rule: Guidelines for Compliance, s Interneta, http://www.nacua.org/nacualert/docs/GLB_Note_051603i.html, 17. svibnja 2011.

[13] LOVREK, I. ET AL: Komunikacijske mreže (radna inačica udžbenika v0.1), Fakultet elektrotehnike i računarstva, Zagreb, Hrvatska, 2010.

[14] Standard for the Format of ARPA Internet Text Messages, IETF RFC 882, s Interneta, http://www.ietf.org/rfc/rfc0822.txt, 18. svibnja 2011.

[15] Internet Message Format, IETF RFC 5322, s Interneta, http://www.ietf.org/rfc/rfc5322.txt, 18. svibnja 2011.

[16] Uniform Resource Identifier (URI): Generic Syntax, IETF RFC 3986, s Interneta, http://www.ietf.org/rfc/rfc3986.txt, 18. svibnja 2011.

[17] The Open Group Base Specifications Issue 6, IEEE Std 1003.1, s Interneta, http://pubs.opengroup.org/onlinepubs/009695399/, 18. svibnja 2011.

[18] ARI: SSH environment – circumvention of restricted shells, Bugtraq, 24. lipnja 2002., s Interneta, http://seclists.org/bugtraq/2002/Jun/312, 19. svibnja 2011.

[19] GARFINKEL, S.; SPAFFORD, G.: Practical UNIX & Internet Security, s Interneta, http://docstore.mik.ua/orelly/networking/puis/, 19. svibnja 2011.

[20] AUGER, R.: Cross Site Scripting, s Interneta, http://projects.webappsec.org/w/page/13246920/Cross-Site-Scripting, 19. svibnja 2011.

[21] COWAN, C. ET AL: Buffer overflows: attacks and defences for the vulnerability of the decade, DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings, vol. 2, Hilton Head, SC, 2000.

[22] Twilight Hack, s Interneta, http://wiibrew.org/w/index.php?title=Twilight_Hack, 19. svibnja 2011.

[23] WHEELER, D. A.: Secure programmer: Countering buffer overflows, s Interneta, http://www.ibm.com/developerworks/linux/library/l-sp4/index.html, 17. svibnja 2011.

[24] STEIN, D. L.; STEWART, J. N.: The World Wide Web Security FAQ, s Interneta, http://www.w3.org/Security/Faq/www-security-faq.html, 17. svibnja 2011.

[25] FERGUSON, R.: Sony faces legal action over attack on PlayStation network, s Interneta, http://www.bbc.co.uk/news/technology-13192359, 19. svibnja 2011.

[26] Hypertext Transfer Protocol – HTTP/1.1, IETF RFC 2616, s Interneta, http://www.ietf.org/rfc/rfc2616.txt, 18. svibnja 2011.

[27] NEWSHAM, T.: Format String Attacks, Bugtraq, 9. rujna 2000., s Interneta, http://seclists.org/bugtraq/2000/Sep/214, 19. svibnja 2011.

[28] ANONYMOUS: Once upon a free()..., Phrack, Volume 0x0b, Issue 0x39, Phile #0x09 of 0x12, s Interneta, http://www.phrack.org/issues.html?issue=57&id=9, 21. svibnja 2011.

[29] Secure Coding Guidelines for the Java Programming Language, Version 3.0, s Interneta, http://www.oracle.com/technetwork/java/seccodeguide-139067.html, 21. svibnja 2011.

[30] NEWHAM, C.: Learning the bash Shell, 3rd edition, s Interneta, http://book.chinaunix.net/special/ebook/oreilly/Learning_bash_Shell/, 21. svibnja 2011.

Sažetak

Pisanje sigurnih izvornih tekstova programa

Sigurnost je potreba raznih vrsta programa, a o njoj se u školama rijetko uči. Programer treba biti svjestan dobre prakse industrije računalne sigurnosti. Ovaj rad neformalno uvodi u normu *Common Criteria* te kroz nju razmatra sigurnosne zahtjeve koje siguran izvorni tekst programa mora zadovoljiti. Opisani su općeniti sigurnosni pristupi i principi, ali i nužne sigurnosne metode poput provjeravanja ulaza, izbjegavanja preliva međuspremnika, opreznog oslanjanja na vanjske resurse i razumnog slanja povratnih informacija. Razmatrani su i problemi specifični programskim jezicima C/C++, Java i skriptnim jezicima ljuske. Zaključno je proveden niz ispitivanja izvornih tekstova dostupnih na fakultetskom Webu.

Ključne riječi: računalna sigurnost, siguran program, siguran izvorni tekst programa, sigurnosni zahtjevi, sigurnosni principi

Abstract

Writing secure code

Security is a requirement of many kinds of programs, but is rarely taught in schools. A programmer must be aware of good practices in the computer security industry. This work informally introduces the Common Criteria standard, through which security requirements that secure program code must satisfy are discussed. General security approaches and principles are described, but also the necessary security methods such as input validation, avoiding buffer overflow, carefully relying on other resources and sending information back sensibly. C/C++, Java and shell scripting language-specific problems are discussed as well. Finally, a series of tests was carried out on source code availiable from the faculty Web.

Keywords: computer security, secure program, secure code, security requirements, security principles

Dodatak A

Izvorni tekstovi programa nad kojima su provedena ispitivanja dohvaćeni su sa sljedećih fakultetskih Web-adresa:

- Konverter: http://www.fer.hr/_download/repository/OR_2010_11_Primjer_za_4._laboratorijsku_vjezbu.zip
- Mojping: http://www.fer.hr/_download/repository/mojping%5B2%5D.html
- Mojdaytime: http://www.fer.hr/_download/repository/mojdaytime.html
- Nweb: http://www.fer.hr/_download/repository/nweb.html