IEGULDĪJUMS TAVĀ NĀKOTNĒ!

Pētījums Nr. 1.20. Interneta lietotāju uzvedības analīzes rīks

Finansēšanas līgums Nr. L-KC-11-0003

Iepirkums Nr. ITKC/2013-1.20RP-3

Testēšanas rokasgrāmata

Rīga, 2013

SIA IT KOMPETENCES CENTRS
Reģistrācijas numurs: 40103326439
Rīgā, Lāčplēša ielā 41, LV-1011
Kontaktpersona:
Dace Skrastiņa
Tālr.: 67844273
Fakss: 67315315
E-pasta adrese: dace.skrastina@lursoft.lv

Satura rādītājs

Tabulu rādītājs

Attēlu rādītājs

1. Ievads

Dokuments "Testēšanas rokasgrāmata" ir daļa ERAF līdzfinansēta projekta "Informācijas un komunikāciju tehnoloģiju kompetences centrs", ko SIA "IT kompetences centrs" īsteno kopā ar nozares un zinātniskajiem sadarbības partneriem īsteno ,pētījuma Nr. 1.20. "Interneta lietotāju uzvedības analīzes rīks" dokumentācijas, ko saskaņā ar iepirkuma Nr. ITKC/2013-1.20RP-3 rezultātā noslēgto līgumu Nr. PL/1.20/2013-2 sagatavoja SIA "Odo". Dokumentā ir aprakstītas pētījumā izstrādātā sistēmas prototipa testēšanas paņēmieni un rīki.

2. Pilnā pārlase un zīmīgie sistēmas stāvokļi

Teorētiski var pieņemt, ka sistēma darbojas pareizi, tikai tad, ja tā ir pārbaudīta visos iespējamajos veidos. Tomēr, ja kā piemēru apskatām vienkāršu kalkulatoru, kas darbojas ar veseliem skaitļiem no -99 līdz 100 un atbalsta četras darbības: pieskatīšanu, atņemšanu, reizināšanu un dalīšanu, tad tajā *A darbība B* ir iespējams ievadīt $4*200^2$ =160000 veidos. Gadījumā, ja iespējams darbības apvienot, piemēram, *2+4*5*, tad iespējamo variantu skaits ir milzīgs.

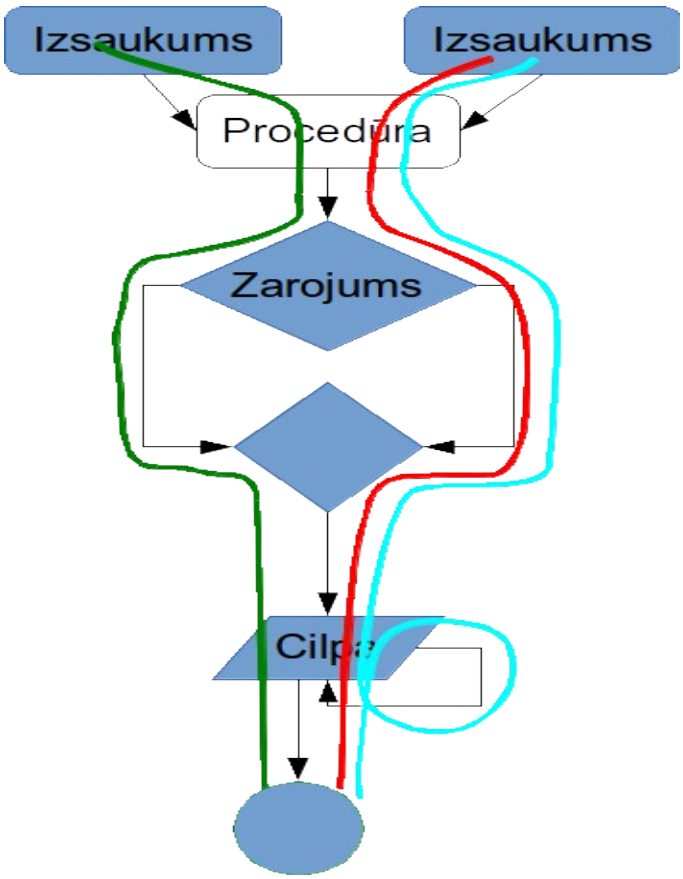
Līdz ar to praksē var pārbaudīt visus iespējamos variantus tikai salīdzinoši vienkāršām sistēmām kurām kopējais stāvokļu skaits nav liels (piemēram, vienkāršiem liftiem, sensoriem u.tml). Praktiski netriviālām sistēmām ļoti bieži nepārbauda *visus* iespējamos stāvokļus, bet tikai t.s. *zīmīgos stāvokļus*. Zīmīgie sistēmas stāvokļi ir tādi, kurā sistēma darbojas principiāli savādāk kā iepriekš apskatītajos.

Raugoties uz sistēmas zemāko realizācijas slāni (programmatūras pirmkodu) tiek uzskatīts, ka ir pārbaudīti visi sistēmas zīmīgie stāvokļi, ja **vismaz vienu reizi ir notestēts**:

1. **Viss programmatūras pirmkods** — izpildot testus, jebkura koda rindiņa ir izsaukta vismaz vienu reizi.
2. **Visi programmatūras zari** — ja programmatūrā ir izvēles un zari, tad visi no tiem ir pārbaudīti),
3. **Visi izsaukuma ceļi** — ja kādu koda daļu var izsaukt no vairākām citām programmas vietām, tad ir pārbaudīti visi iespējamie izsaukuma veidi,
4. **Visas programmas cilpas** — ja programmā ir cikli un cilpas, tad katra no tām ir izsaukta vismaz vairāk par vienu reizi.

Programmatūras zaru un cilpu dēļ praksē parasti sanāk tā, ka ir programmas vietas, kas tiek notestētas vairāk par vienu reizi, jo lai notestētu visus zarus un cilpas, tad kāda no programmas kopīgajām daļām ir jāizpilda vairākkārt. Sekojošā piemērā, lai vismaz vienu reizi pārbaudītu visus izsaukumu veidus, zarojumus un cilpas, procedūra ir jāizsauc *vismaz* trīs reizes. Un arī tad pārklājums nav pilnīgs, jo no viena izsaukuma nepārbauda cilpas darbību, bet no otra izsaukuma nepārbauda abus zarošanās nosacījumus. Tāpēc precīzākam testēšanas rezultātam, augšminētos četrus nosacījumus var papildināt ar vēl vienu:

5. pārbaudīt **augšminēto nosacījumu visas iespējamās kombinācijas**.



Attēls 1: Minimālais programmas izpildes pārklājums

Ja sistēmu testē caur lietotāja saskarni, tad zīmīgo gadījumu noteikšana nav algoritmiski precīzi aprakstāma. Tomēr arī tādā gadījumā var noteikt galvenos zīmīgos stāvokļus, kas parasti ir t.s. "biznesa lietojumi" (ang. *business case*), jeb piemēri.

Biznesa lietojumu piemēri ir gadījumi, kādi ir iespējamie sistēmā reģistrēto klientu stāvokļi (jauns, apstiprināts, neaktīvs u.tml.), dažādu datu ievades secība, u.tml.

3. CRUD testēšanas paņēmiens datu objektiem

Sistēmā, kurā tiek apstrādāti un saglabāti dati, viena no sarežģītākajām risināmajām problēmām ir testu rezultātu atkarība no sistēmā saglabātajiem datiem. Piemēram, meklējot lietotāju "*Jānis Bērziņš*", atkarībā no sistēmas datu kopas par pozitīvs rezultāts var būt ļoti dažāds:

1. Ja "Jānis Bērziņš" ir unikāls lietotājs un sistēmā ir šāds lietotājs, tiek atrasts *viens* lietotājs,
2. Ja sistēma nav lietotāja "Jānis Bērziņš", tad netiek atrasts *neviens* atbilstošs lietotājs,
3. Ja sistēmā ir vairāki lietotāji "Jānis Bērziņš", jo vārds un uzvārds nav pietiekami unikāls meklēšanas nosacījums, tiek atrasti *vairāki* lietotāji.

Šādu datu atkarību dēļ, sistēmas testēšana iepriekš noteiktā algoritmiskā veidā ir sarežģīta un bieži vien pat neiespējama. Sarežģītās sistēmās, kur viena datu kopa ir atkarīga no citām datu kopām, veic īpašu testa datu kopu pārvaldību, pārliecinoties, ka sistēmas dati atbilst testos paredzētajam stāvoklim (piemēram, veicot datu bāzes importu no iepriekš sagatavota eksporta faila pirms testiem, u.tml.).

Tomēr bieži vien ir iespējams veikt efektīvu testēšanu arī daudz vienkāršākā viedā, izmantojot t.s. CRUD (ang. *Create, Read, Update, Delete*), jeb **Izveidot, Lasīt, Atjaunot, Dzēst** darbības. Atsevišķos gadījumos kāda no darbībām gala lietotājam var nebūt atļauta/realizēta. Piemēram, lietotājam var nebūt tiesības radīt jaunus, vai dzēst esošos datus, vai arī ir atļauts veidot jaunus, bet nav ļauts tos vēlāk mainīt.

Katru no CRUD operācijām var izpildīt līdz galam, vai arī atcelt (ja tas ir atļauts). Līdz ar to, vispārīgā gadījumā, visām datu vienībām ir nepieciešams pārbaudīt sekojošas darbības:

N.p.k.	Darbība	Rezultāts
1.	Izveido jaunu datu objektu un *atceļ* tā izveidi	Objekts netiek izveidots
2.	Izveido jaunu datu objektu un *apstiprina* tā izveidi	Tiek izveidots jauns datu objekts
3.	Atlasa (ar dažādiem kritērijiem) konkrētu datu objektu un *atceļ* tā meklēšanu	Datu objekts netiek atgriezts
4.	Atlasa (ar dažādiem kritērijiem) konkrētu datu objektu un *apstiprina* tā meklēšanu	Tiek atgriezts viens vai vairāki datu objekti
5.	Izmaina datu objekta īpašību vērtības un *atceļ* izmaiņas	Datu objekta vērtības netiek izmainītas
6.	Izmaina datu objekta īpašību vērtības un *apstiprina* izmaiņas	Datu objektā tiek saglabātas jaunās vērtības
7.	Dzēš izvēlēto objektu un *atceļ* dzēšanu	Datu objekts netiek izdzēsts
8.	Dzēš izvēlēto objektu un *apstiprina* dzēšanu	Datu objekts tiek izdzēsts

Tabula 1: CRUD darbību un paredzamo rezultātu apraksts

Pielietojot CRUD principu, testu scenārijos ir jāzina tikai tas, kāds jauns datu objekts tika izveidots, un visas citas darbības tiek veiktas ar jaunizveidoto objektu. Tādējādi var būtiski samazināta testu atkarību no sistēmā esošajiem datiem. Piedevām, ja datu objektam ir atļauts veikt visas četras CRUD darbības, tad pēc veiksmīgi pabeigtiem testiem sistēmas datos vispār nav nekādu izmaiņu, jo visi no jauna radītie dati tiek izdzēsti.

Reālos pielietojumos parasti ir sarežģītākas datu atkarības, piemēram, lai pārbaudītu jauna lietotāju profila statusu, tas ir jāizmanto lietotāja profilā, un pēc tam to vairs nevar izdzēst. Tāpēc nereti ir nepieciešams izveidot vairākus viena veida datu objektus, no kuriem viens vai vairāki tiek izmantoti tālākos testēšanas scenārijos, bet viens tiek izdzēsts. Atkarībā no testēšanas apjoma, var būt nepieciešamība pārbaudīt, lai izmantotos datu objektus vairs nav iespējams izdzēst.

4. Pozitīvā un negatīvā testēšana

Pārbaudot sistēmas darbību, atkarībā no tai nodotajiem/ievadītajiem datiem un sagaidāmajiem rezultātiem, var izdalīt divas atsevišķas testu grupas: pozitīvos un negatīvos testus.

- *Pozitīvajos* testos pārbauda, ka sistēmā *ievadot pareizus* datus, tā *strādā* tā, kā ir paredzēts. Veicot šos testus ievēro sistēmas dokumentācijā aprakstītās prasības un darbību secību un sagaida, ka sistēma darbosies tā, kā ir paredzēts dokumentācijā.
- *Negatīvajos* testos pārbauda, ka sistēmā *ievadot nepareizus* datus, tā *nedarbojas*. Atkarībā no prasībām sistēmas lietojamībai un draudzīgumam, sistēmas nedarbošanās var būt dažāda: sākot ar ievadīto datu ignorēšanu, un beidzot ar dažādiem sistēmas ziņojumiem par to, kas īsti, pēc sistēmas domām, nav pareizi.

Pozitīvos testus izmanto, lai pierādītu, ka ar sistēmu ir iespējams veikt tajā paredzētās darbības. Līdz ar to, standarta pozitīvie testi ir sistēmas lietojumu piemēri, kuros veic visos biznesa procesos paredzētās darbības.

Negatīvos testus izmanto, lai pārbaudītu sistēmas noturību pret nekvalitatīviem vai ļaunprātīgi ievadītiem datiem. Standarta negatīvo testu piemēri ir nenorādītas obligātās vērtības, nepareizi datu formāti (datumu vērtībām) vai diapazoni (skaitliskām vērtībām), neunikālas datu atslēgas, u.tml.

N.p.k.	**Testa veids**	**Paredzamais rezultāts**
1.	Pozitīvais tests ievada *atbilstošus* datus	Sistēma *strādā*, kā paredzēts
2.	Negatīvais tests ievada *neatbilstošus* datus	Sistēma *nestrādā*, kā paredzēts

Tabula 2: Pozitīvie un negatīvie testi un to rezultāti

5. Testu rezultāti un patiesā sistēmas darbība

5.1. Patiesas un viltus veiksmes un neveiksmes

Pēc sistēmas testa izpildes tiek iegūts kaut kāds rezultāts, kam būtu jānorāda:

- ja tests netika izpildīts, sistēma nestrādā pareizi,
- un otrādi — ja tests tika izpildīts, sistēma strādā pareizi.

Tomēr, kā jau tika aprakstīts 2. nodaļā, netriviālām sistēmām praktiski nav iespējams pārbaudīt visus iespējamos stāvokļus, bet t.s. zīmīgo stāvokļu noteikšana nav algoritmiski precīza. Tāpēc ir iespējami gadījumi, kad testu rezultāti neatspoguļo precīzi sistēmas patieso darbību:

- testa rezultāts liek domāt ka sistēma strādā pareizi, lai gan tā nestrādā pareizi,
- vai arī otrādi — testa rezultāts liek domāt, ka sistēma nestrādā pareizi, lai gan patiesībā tā strādā pareizi.

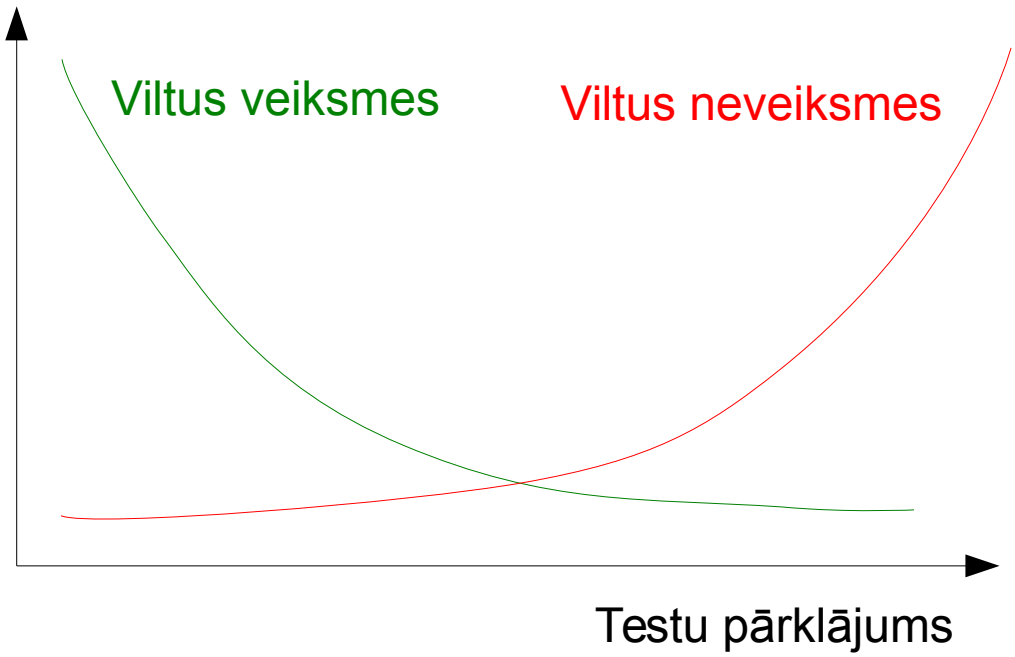
Piemēram, lietotāju autentifikācijā var būt gadījumi, kad ievadot nepareizu vārdu un paroli, tomēr izdodas pieteikties sistēmā, kas *viltus veiksme* (ang. *false positive*). Bet var būt arī tā, ka, pat ievadot pareizu vārdu un paroli sistēmā neizdodas pieteikties, kas ir *viltus neveiksme* (ang. *false negative*).

N.p.k.	Testa rezultāts	Rezultāta apraksts	Patiesais sistēmas stāvoklis
1.	Pozitīvais tests izgājis veiksmīgi	Patiesa veiksme	Labs
2.	Negatīvais tests izgājis neveiksmīgi	Patiesa neveiksme	Labs
3.	Pozitīvais tests izgājis neveiksmīgi	Viltus neveiksme	Slikts
4.	Negatīvais tests izgājis veiksmīgi	Viltus veiksme	Slikts

Tabula 3: Patiesu un viltus rezultātu iespējamie varianti

Sistēmu nepilnīgi testējot, ir iespējams, ka kļūdas netiek atklātas, lai gan tās patiesībā ir. Tas nozīmē, ka ir daudz viltus veiksmju. Atklājot arvien vairāk iespējamo kļūdu, samazinās viltus veiksmju gadījumi. Tāpēc parasti ir nepieciešams maksimizēt t.s. *testu pārklājumu* izstrādājot arvien plašākus un detalizētākus testus.

Tomēr praktiski arvien lielāks testu pārklājums prasa arvien lielāku testu uzturēšanas darbu, tāpēc testi var būt neatbilstoši sistēmas jaunākajai versijai vai datu kopai. Līdz ar to, arvien vairāk pieaug iespēja, ka testējot sistēmu, tiek atklātas viltus neveiksmes. Līdz ar to, no darbietilpības optimālā testu kopa ir tāda, kur, gan viltus veiksmju, gan viltus neveiksmju ir pēc iespējas mazāk.



Attēls 2: Viltus veiksmju un neveiksmju atkarība no testu pārklājuma

Atkarībā no tā, cik ir viltus veiksmju un neveiksmju, var pieņemt lēmumu, kā tālāk attīstīt sistēmas testēšanu.

5.2. Regresijas testi

Teorētiski, veicot pat minimālu sistēmas izmaiņu, jaunā sistēma vairs nav tāda pati un visi testi, kas tika veikti iepriekšējai sistēmas versijai, vairs nav spēkā jaunajai versijai. Praktiski, protams, var šķirot triviālas izmaiņas (lauku izklājums uz ekrāna) no būtiskām (biznesa loģiku) mainošām izmaiņām. Tāpēc ne visām sistēmas izmaiņām ir nepieciešams veikt vienādi pilnīgu (vienāda pārklājuma) testēšanu. Praktiski, pēc sistēmas izmaiņām, tiek testēta tikai tā sistēmas daļa, kas tika mainīta, cerot un paļaujoties uz to, ka sistēma ir pietiekami modulāra un izmaiņas vienā vietā būtiski neietekmē citas sistēmas daļas.

Tomēr, var gadīties tā, ka uzlabojot vienu sistēmas daļu, izmaiņu rezultātā kāda cita sistēmas daļa vairs nestrādā nemaz, vai strādā sliktāk. Tāpēc sistēmā ir nepieciešams veikt vispārīgu testēšanu, ar kuru nepārbauda pilnīgi visu sistēmu, bet tikai pārliecinās, ka *sistēma nav palikusi sliktāka*. Šādus "nepilnīgus" testus, ar kuriem galvenokārt nosaka tikai to, vai sistēma nav palikusi sliktāka, sauc par *regresijas testiem*. Tā kā regresijas testus nākas regulāri atkārtot, šos testus visbiežāk cenšas automatizēt, izmantojot īpašas programmas, jeb automātiskus testēšanas rīkus.

Modernās izstrādes vidēs automātiski testēšanas rīki ir iekļauti standarta komplektā, piemēram Java, vadošais ir Junit rīks. Ar šiem rīkiem var pārbaudīt atsevišķu sistēmas komponenšu darbību, un dažkārt arī sistēmas moduļu darbību.

Tīmekļa bāzētām sistēmām, kā gala lietotāja aizstājēju izmanto **Selenium IDE** rīku, kas ir Firefox pārlūkprogrammas spraudnis vai arī Selenium Server, kas, izmanto Firefox pārlūkprogrammas API, bet darbojas Eclipse izstrādes vidē.

Tālāk dokumentā ir apskatītas tehniskās detaļas, kas ir svarīgas tīmekļa lietotņu testēšanā.

6. HTTP vispārīgs apraksts

Tīmekļa pārlūkprogramma un serveris savstarpēji sazinās, izveidojot TCP savienojumu. Senāk katram HTTP pieprasījumam izmantoja atsevišķu TCP savienojumu, bet sākot ar HTTP/1.1 viena savienojuma ietvaros var veikt vairākus pieprasījumus, ja pārlūkprogramma nosūta *keepalive* parametru. HTTP klients (pārlūkprogramma) nosūta teksta pieprasījumu serverim, bet serveris atgriež tīmekļa lapas saturu pamatā HTML formā (vai citā formātā, piemēram, CSS, JavaSript, vai failu MIME formātā).

6.1. GET pieprasījums

Get pieprasījums ir izplatītākais tīmekļa pieprasījumu veids, kuru parasti izmanto pastāvīgi pieejamu resursu aplūkošanai.

```
telnet odo.lv 80
```

ar šo komandu tiek izveidots savienojums ar serveri, piemēram, ar serveri odo.lv . Pārliecinās, ka parādās paziņojums

```
Trying 92.240.68.210...
Connected to odo.
Escape character is '^]'.
```

Ievada komandu:

```
GET / HTTP/1.0
```

Pārliecinās, ka tiek atgriezts lapas saturs HTML formātā.

6.2. POST pieprasījums

Līdzīga kā iepriekš, kad ar telnet komandu ir izveidots TCP savienojums, ievada pieprasījumu:

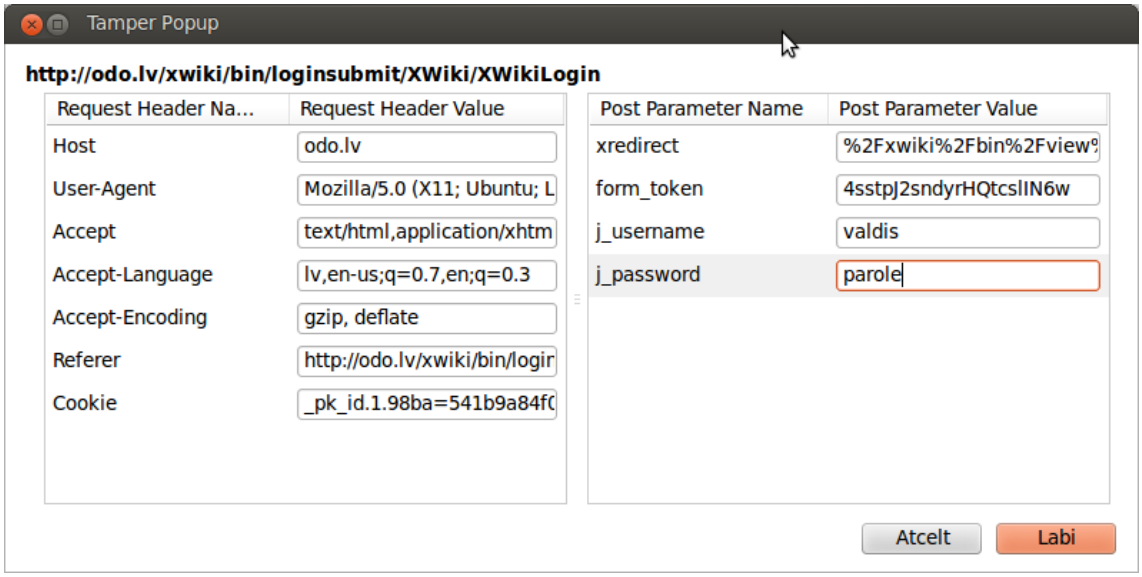
```
POST / HTTP/1.0
Host: odo.lv
Content-Type: application/x-www-form-urlencoded
Content-Length: 11

user=Valdis
```

un spiež Enter

Content-Length ir jābūt norādītam sagaidāmajam simbolu (burtu) skaitam!

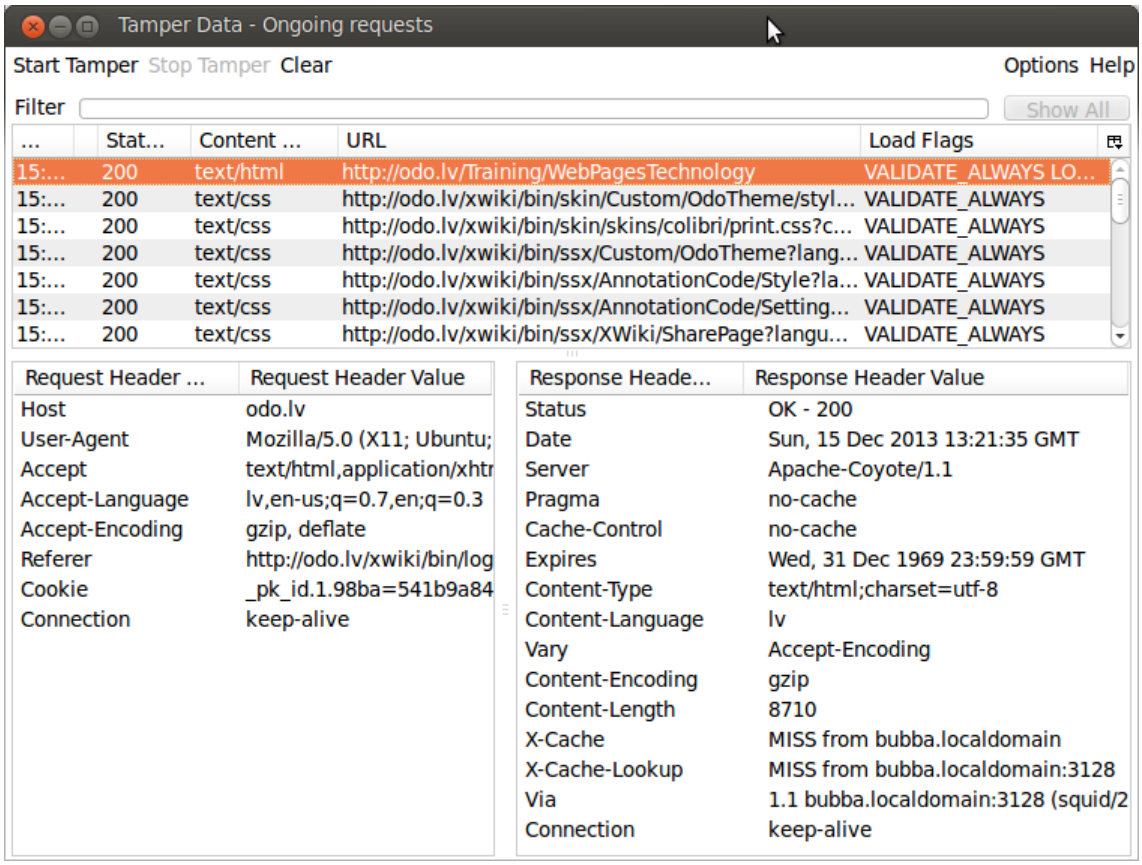
Visa informācija, kas turpinās līdz tukšajai rindai ir t.s. pieprasījuma *iesākums* jeb *hederis,* kurā var būt norādīti vairāki *parametri* (*Host, Content-Type, Content-Length*) .

Lai arī visu pārlūkprogrammas uzvedību var emulēt manuāli ar telnet programmu, praktiski to novērot un, nepieciešamības gadījumā mainīt ir vieglāk, izmantojot pārlūkprogrammas spraudņus, piemēram, **Tamperdata**.



Attēls 3: HTTP pieprasījuma hederi un POST parametru vērtības Tamperdata spraudnī

Izmantojot pieprasījuma hederu parametrus, pārlūkprogramma servrim vai otrādi, var nodod dažādu papildu informāciju. Sekojošajā attēlā ir parādīti pārlūkprogrammas nosūtītie (pieprasījuma) hedera parametri un tīmekļa servera atgrieztie hedera parametri Tamperdata spraudņa logā:



Attēls 4: Pieprasījuma un atbildes hedera parametri Tamperdata spraudnī

Novērojot tīmekļa pārlūkprogrammas un tīmekļa servera saziņu, var pārliecināties, ka viena lietotājam parādīta tīmekļa lapa patiesībā var būt sadalīta daudzos atsevišķos

resursos (atsevišķos failos lapas noformējumam un stilam, attēliem, u.t.t.).

7. Tīmekļa sistēmu drošība

7.1. Sesijas un sīkfaili

Vienkāršs tīmekļa (HTTP) serveris, kas nodrošina parastu (GET/POST) pieprasījumu apstrādi, ir bezstāvokļa (ang. *stateless*), jeb vienkārši sakot, tas neko neatceras. Ja lietotājam ir nepieciešams atcerēties kaut kādu stāvokli (piemēram, to, ka lietotājas ir autentificēts), tad tas tiek saglabāts pārlūkprogrammā t.s. sīkfailā (*cookie*). Katrā pieprasījuma reizē pārlūkprogramma bez pieprasītajiem datiem pieprasījumam pievieno arī visu sīkfailos saglabāto informāciju. Atkarībā no tā, kāda informācija no sīkfailiem tiek nodota, serveris ļauj vai neļauj rādīt kādus tīmekļa resursus, u.tml.

Sīkfailu drošība ir atkarīga no lietotāja izmantotās pārlūkprogrammas un operētājsistēmas. Piemēram, ja lietotājs ir padarījis par koplietojamu savu mapi, tad cits tīkla lietotājs var nokopēt pārlūkprogrammas sīkfailus un uzdoties par citu lietotāju. Daļēji šo problēmu risina, ieviešot sesijas sīkfailus (drīzāk, "cepumus", jo tie nemaz netiek saglabāti failā), kurus glabā tikai pārlūkprogrammas operatīvajā atmiņā. Otra problēma, kas saistīta ar sīkfailiem ir tā, ka sarežģītās tīmekļa lietotnēs ir nepieciešams uzglabāt daudz un dažādu informāciju, un tā visa ir atkārtoti jāpārsūta veicot katru tīmekļa pieprasījumu.

Sarežģītās tīmekļa lietotnēs, izmanto t.s. lietotņu serverus (ang. *application server*), kuri par katru lietotāju saglabā kaut kādu informāciju (lietotāja stāvokli) pie sevis. T.i. lietojumu serveris nav pilnībā bezstāvokļa serveris. Konkrētā lietotāja stāvokli lietojumu serveri saglabā t.s. sesijā (ang. *session*), kura nosaka ar unikālu, varbūtīgi ģenerētu, skaitli. Tad tīmekļa pārlūkprogramma sīkfailā saglabā tikai sesijas identifikatoru (bieži vien to sauc par *sessionid*), kuru izmanto kā atslēgu, lai izmantotu uz servera saglabātā stāvokļa (ko parasti sauc par kontekstu) noteikšanai. Praktisku/drošības apsvērumu nolūkos bez sesijas identifikatora pārlūkprogrammā saglabā arī vēl dažus citus parametrus (piemēram, hešotu lietotāja vārdu un paroli), kuru izmanto sesijas atbilstības pārbaudei.

Cookies ▾ Filter ▾ Default (Accept cookies) ▾

Name	Value	Domain	Raw Size	Path	Expires	HttpOnly
⊞ jforumAutoLogin	1	odo.lv	16 B	/	piektdiena, 2014. gada 5. septembris, plkst. 17 un 16	
⊞ jforumUserHash	68ec00336158009bf53992e15c528dff	odo.lv	46 B	/	piektdiena, 2014. gada 5. septembris, plkst. 17 un 16	
⊞ jforumUserId	3	odo.lv	13 B	/	piektdiena, 2014. gada 5. septembris, plkst. 17 un 16	
⊞ JSESSIONID	F2DA52ADCEA2039F22A04F73CC873543	odo.lv	42 B	/	Session	HttpOnly
⊞ language	lv	odo.lv	10 B	/	trešdiena, 2023. gada 13. decembris, plkst. 14 un 49	
⊞ _pk_ref.1.98ba	["","",1387111784,"http...476/linux-ws-win	odo.lv	123 B	/	pirmdiena, 2014. gada 16. jūnijs, plkst. 03 un 49	
⊞ _pk_id.1.98ba	541b9a84f0f6a0b4.137770....1387111784	odo.lv	67 B	/	otrdiena, 2015. gada 15. decembris, plkst. 14 un 49	
⊞ _pk_ses.1.98ba	*	odo.lv	15 B	/	svētdiena, 2013. gada 15. decembris, plkst. 15 un 19	

Attēls 5: Sīkfaili Firebug spraudņa logā

Attēlā attēloti sīkfaili Firebug spraudņa logā. No tiem melnā krāsā ir attēloti sīkfaili, kuri tik tiešām tiek saglabāti failā, un zaļā krāsā ir "virtuāls sīkfails", kura glabā pārlūkprogrammas operatīvajā atmiņā.

7.2. SQL injekcijas

Veidojot tīmekļa lietotni, bieži nākas veidot vaicājumus (pieprasījumus), kuru konkrētā vērtība ir atkarīga no lietotāja, sistēmas stāvokļa u.tml. Vienkāršs (t.s. naivais) veids, šādu pieprasījumu izveides paņēmiens, ir interpretēt ievadītos parametrus kā izpildāmas komandas (kodu) attiecīgā lietotnes infrastruktūrā.

Piemēram, lai atlasītu datus par tekošo lietotāju, var dinamiski (programmas kodā) veidot SQL vaicājumu:

```
statement = "SELECT * FROM users WHERE name = '" + userName + "';"
```
Šādā dinamiski veidotā vaicājumā tekošā lietotāja vārda vietā kāds var ievadīt saturu, kurš atceļ iepriekš sagatavoto vaicājumu, un papildina to ar patvaļīgu:

```
' or '1'='1
' or '1'='1';/*'
'; SELECT * FROM passwords;'
1;DROP TABLE users
```
Ar SQL injekcijām cīnās dažādos veidos:

1. Nofiltrējot izņēmuma simbolus
2. Pārbaudot datu tipus
3. Parametrizējot vaicājumu izveidi, piemēram, Java programmēšanas valodā, izmantojot **java.sql** pakotni:

```
prep = conn.prepareStatement ("SELECT * FROM USERS WHERE USERNAME=? AND PASSWORD=?");
prep.setString(1, username);
prep.setString(2, password);
prep.executeQuery();
```

7.3. Shell injekcijas

Ja no lietojumprogrammas tiek izsaukta sistēmas čaulas komanda, tad, brīvi interpretējot tajā nodotos parametrus, var injicēt arī sistēmas čaulas (piemēram, Bash vai Windows CLI) komandas.

Piemēram, ja no PHP koda tiek izsaukta komanda:

```
<?php
passthru("/bin/funnytext " . $_GET['USER_INPUT']);
?>
```
Ļaunprātīgs lietotājs var norādīt

```
; slikta_komanda
```
Ar semikolu pabeidzot iepriekšējo komandu un turpinot jaunu komandu aiz tā.

7.4. Cita veida injekcijas

Līdzīgi iepriekš aprakstītajam, ļaunprātīgi lietotāji var iekļaut ļaunprātīgu HTML kodu (piemēram, lapas komentāros) kas parāda saturu no citas tīmekļa vietnes. Bez HTML koda tīmekļa lietotnēs var iekļaut no citas vietas arī JavaScript programmu kodu. Nereti šādu darbību veic apzināti, piemēram, sociālās vietnēs, vai ziņu portālos, kuri sadarbojas lietotāju uzvedības pētīšanā, lai šīs ziņas piedāvātu reklāmdevējiem.

Tīmekļa vietnēs, kas neatbalsta, piemēram, HTML vai JavaScript koda izsaukšanu no citas tīmekļa vietnes, parasti katram pierasījumam ģenerē unikālu identifikatoru, kura vērtību pārbauda ar Get vai Post pierprasījumu nodotajiem datiem. Vairāk par to skatīt:

- http://en.wikipedia.org/wiki/Code_injection
- http://en.wikipedia.org/wiki/Csrf

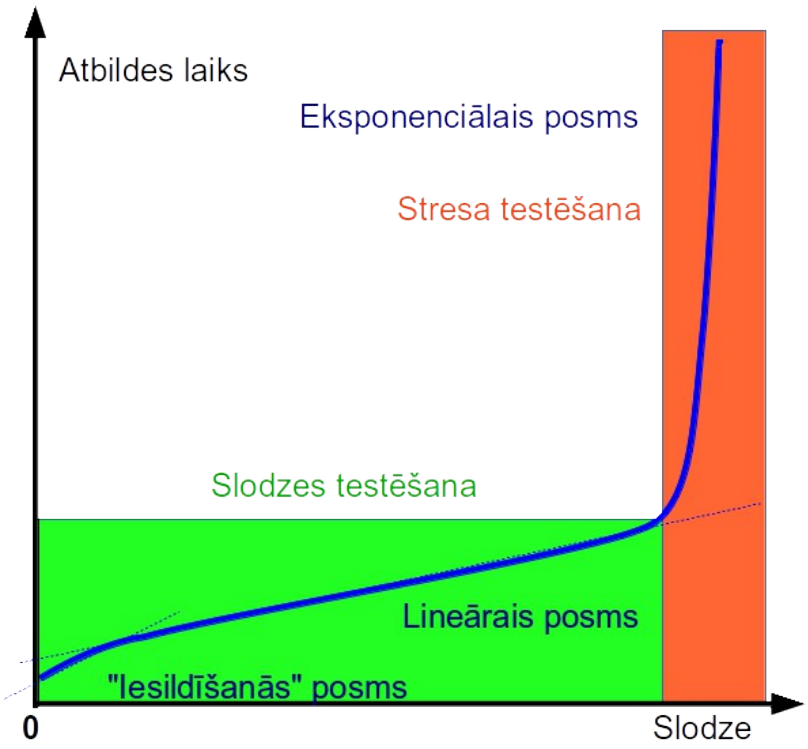
8. Sistēmas veiktspēja

Tā kā tīmekļa lietotnes pēc definīcijas ir daudzlietotāju sistēmas, jeb tās ir paredzētas vairākiem vienlaicīgiem lietotājiem, tad ir nepieciešams noteikt, cik daudz lietotāju sistēma spēj izturēt.

Veiktspējas testēšanā izdala divus atšķirīgus testu veidus:

1. **Slodzes testēšana** (ang. *load testing*) ir zināmā mērā pozitīvais tests, kurā nosaka, ar cik lielu slodzi sistēma spēj tikt galā. Ar slodzes testiem vienmērīgi pieaudzē sistēmas slodzi un vēro, kā izmainās veiktspējas rādītāji: sistēmas slodze, izmantotais procesora laiks un atmiņa, un servera atgrieztā satura laiks.
2. **Stresa testēšana** (ang. *stress testing*) ir sistēmas testēšana ekstremālos apstākļos. Šādā testā sistēma parasti pāriet no interaktīvas datu apstrādes uz rindu apstrādi, un, ja stresa slodze ir īslaicīga, tā var atgriezties sākotnējā stāvoklī, pēc tam, kad rindā stāvošie uzdevumi ir izpildīti. Ja stresa slodze ir ilgstoša, tad sistēma var palikt tik lēna, ka ir praktiski uzskatāma par apstājušos, vai arī tiek pārsniegts kāds no sistēmas fiziskajiem ierobežojumiem, piemēram, operatīvās un maiņvietas apgabala kopīgā atmiņa, un tā pārstāj darboties.

Tā kā sistēmā ir dažādas aiztures (latentums), ko nosaka datu pārraide tīklā, datu lasīšana un rakstīšana diskā, procesora laika izdalīšana sistēmas procesiem u.tml., tā nespēj momentā atbildēt pat tad, ja tai nepastāv nekāda slodze. Tāpēc pat bez jebkādas slodzes pastāv kāda galīga laika nobīde (parasti dažas/daži desmiti milisekundes) kādā sistēma spēj apkalpot lietotāju pieprasījumus. Pēc tam kādu laiku sistēmas veiktspēja parasti ir proporcionāli slodzes lielumam — jo vairāk ir pieprasījumu sistēmai, jo proporcionāli ilgāk tā veic pieprasījumu apstrādi. Sistēmas slodzes testēšanu veic šajā sistēmas veiktspējas posmā. Pēc tam, kad tiek sasniegti sistēmas fiziskie limiti (piemēram, procesors, disks vai atmiņa ir 100% noslogots/aizņemts) tad neliels slodzes pieaugums būtiski pasliktina sistēmas veiktspēju. Šajā posmā veic sistēmas stresa testēšanu. Tad sistēma vairs nespēj apstrādāt visus pieprasījumus interaktīvi un veicamie uzdevumi uzkrājas rindās, un kamēr rindas nav pārpildītas (to parasti nosaka pieejamais operatīvajās atmiņas un maiņvietas apgabala daudzums), tad sistēma pēc slodzes samazināšanās spēj atgriezties normālā darba režīmā. Pretējā gadījumā sistēma "uzkaras" vai paliek tik lēna (tā kā disks ir ~1k reizes lēnāks par operatīvo atmiņu) ka nav iespējams sagaidīt tās atgriešanos normālā darba režīmā.



Zīmējums 1: Sistēmas darbības atkarība no slodzes

Lai noteiktu, cik lielu slodzi testējamā sistēma spēj izturēt, pirmkārt ir nepieciešams vērot sistēmas darbību.

Vienkāršākais un pieejamākais rīks ir Unix-veidīgās sistēmās (Linux, Mac, AIX, HP-UX, BSD u.tml.) ir **top** rīks:

```
top - 16:25:27 up 268 days, 23:30,  1 user,  load average: 0.01, 0.17, 0.25
Tasks: 122 total,   2 running, 119 sleeping,   0 stopped,   1 zombie
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8177104k total,  7938824k used,   238280k free,  1086664k buffers
Swap:  3895756k total,    36364k used,  3859392k free,  4672596k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    1 root      20   0 24320 1884 1096 S    0  0.0   0:57.92 init
    2 root      20   0     0    0    0 S    0  0.0   0:02.27 kthreadd
    3 root      20   0     0    0    0 S    0  0.0  10:20.44 ksoftirqd/0
    6 root      RT   0     0    0    0 S    0  0.0   0:24.00 migration/0
    7 root      RT   0     0    0    0 S    0  0.0   0:49.24 watchdog/0
    8 root      RT   0     0    0    0 S    0  0.0   1:19.35 migration/1
   10 root      20   0     0    0    0 S    0  0.0   6:05.89 ksoftirqd/1
   12 root      RT   0     0    0    0 S    0  0.0   0:39.15 watchdog/1
   13 root      RT   0     0    0    0 S    0  0.0   0:30.35 migration/2
   15 root      20   0     0    0    0 S    0  0.0   8:25.07 ksoftirqd/2
   16 root      RT   0     0    0    0 S    0  0.0   0:44.63 watchdog/2
   17 root      RT   0     0    0    0 S    0  0.0   0:58.98 migration/3
   19 root      20   0     0    0    0 S    0  0.0   6:08.58 ksoftirqd/3
```

Attēls 6: top komandas izdruka

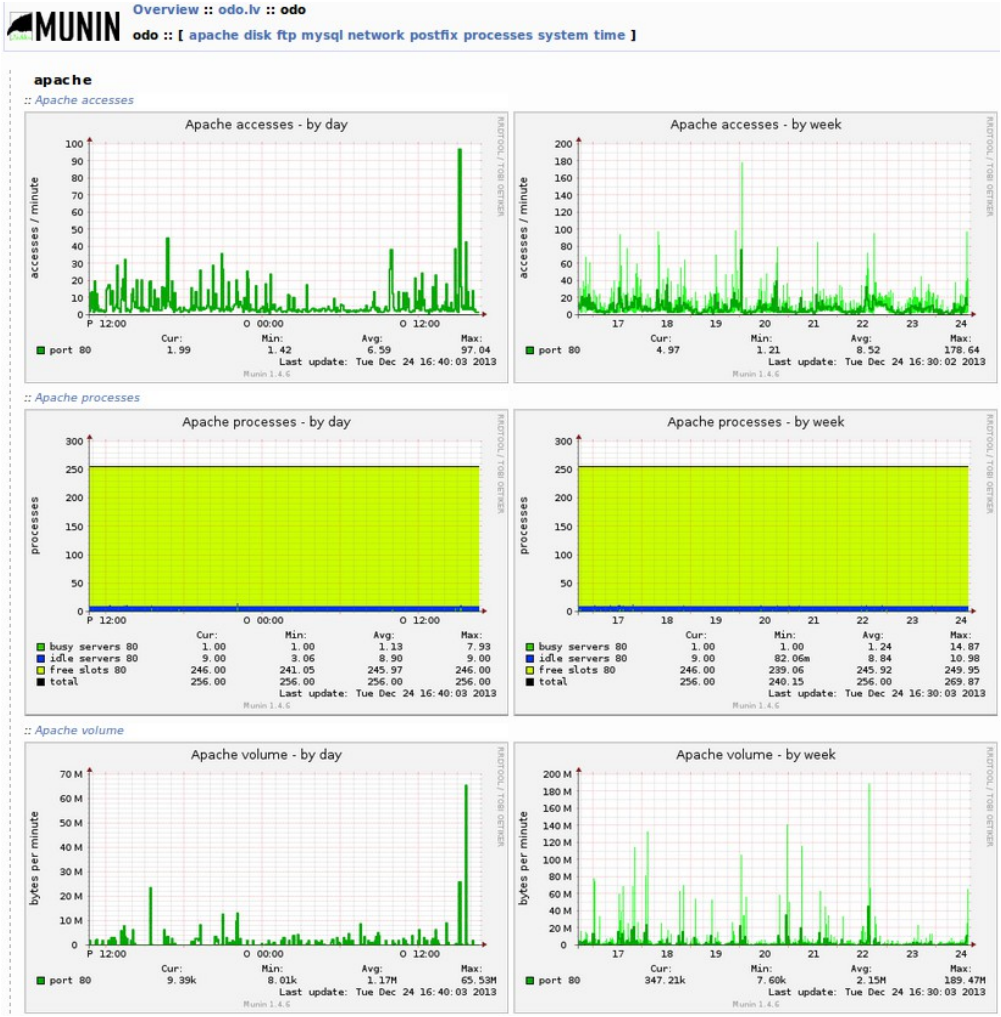
Svarīgākie top komandas radītāji ir:

1. **load average: 0.01, 0.17, 0.25**
   ir uzdevumu, skaits, kurus būtu jāpilda uzreiz, bet tie stāv rindā. Mērījumi tiek uzrādīti par pēdējo minūti, pēdējās piecām un 15 minūtēm. Šiem skaitļiem būtu

jābūt tuvu nullei, vai arī mazāk par 1 uz katru procesoru. Ja skaitli ir lielāki

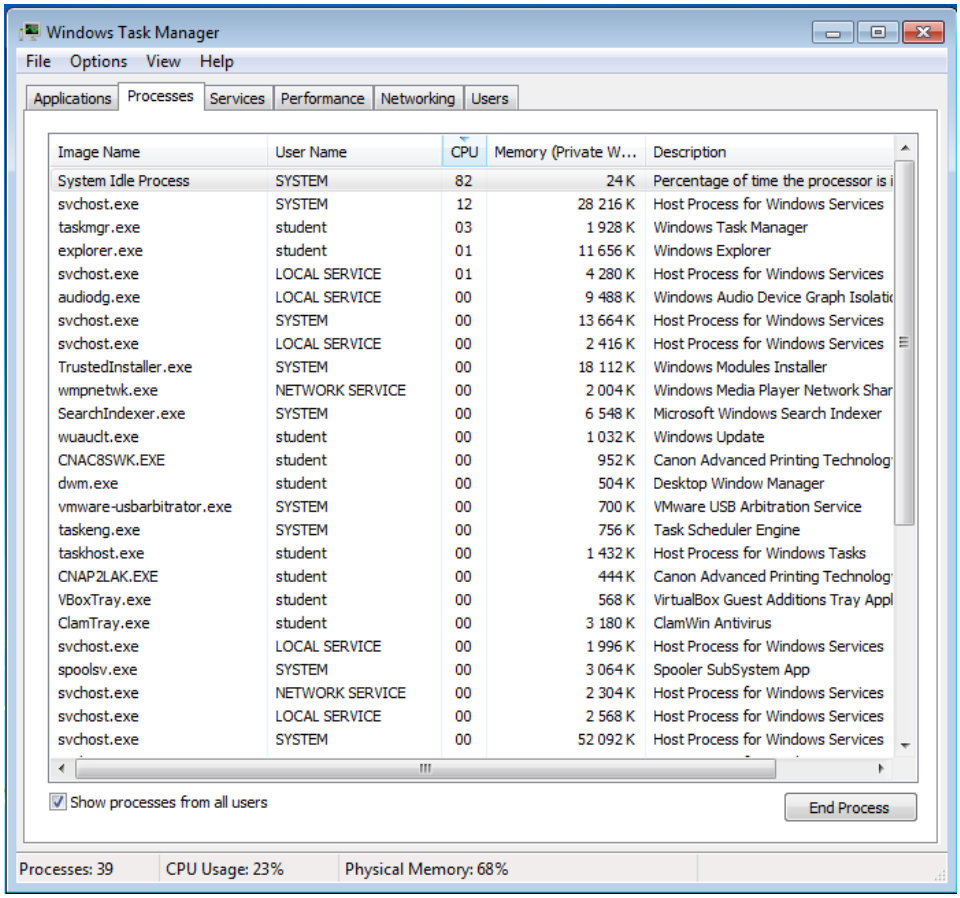
2. ilgstoši, sistēma strādā nevis interaktīvā, bet gan rindu režīmā.
3. **Cpu(s): 1.4%us, 0.2%sy, 0.0%ni, 98.3%id, 0.1%wa, 0.0%hi ...**
   Cik izmanto sistēmas procesoru lietotnes (us), operētājsistēma (sy), procesi ar zemu prioritāti (ni), dīkstāve (id), gaida uz lēnām iekārtām (disku/tīklu, wa), augstas prioritātes procesi (hi) u.c.
4. **Mem: 8177104k total, 7937976k used, 239128k free, 1084504k buffers**
   sistēmā izmantotā operatīvā atmiņa: kopējā (total), cik no tās izmanto operētājsistēma un lietotnes (used) un cik no atlikuma neizmanto (free), un cik izmanto diska rakstīšanas buferēšanai kešošanai (buffers)
5. **Swap: 3895756k total, 36368k used, 3859388k free, 4652676k cached**
   Maiņvietas apgabals kopā (total), izmantots (used), brīvs (free) kešots operatīvajā atmiņā (cached).

Produkcijas sistēmā tiek rekomendēts izmantot plašākus sistēmas veiktspējas rīkus, kā piemēram, Munin



Attēls 7: Munin izdrukas piemērs

Lietojot Windows, vienkāršu sistēmas veiktspējas uzskaiti var veikt, lietojot **Windows Task Manager** (uzdevumu pārvaldnieku).



Attēls 8: Windows Task Manager izdruka

8.1. Sistēmas slodzes radīšana

Ja sistēmai nav paredzēti daudzi tūkstoši vienlaicīgu lietotāju, tad praktiskiem nolūkiem, sistēmas veiktspēju var novērtēt, veicot sistēmas demonstrācija un lietotāju apmācība datorklasē. Tomēr šādi pasākumi ir sarežģīti un laikietilpīgi un plašāk lietojamām sistēmām tas parasti nav iespējams. Tāpēc testēšanai ir vērts izmantot automātiskus rīkus. Tālāk rokasgrāmatā ir aprakstīts, kādi ir pieejami bezmaksas atvērtā pirmkoda rīki, kurus var izmantot tīmekļa sistēmu testēšanā.

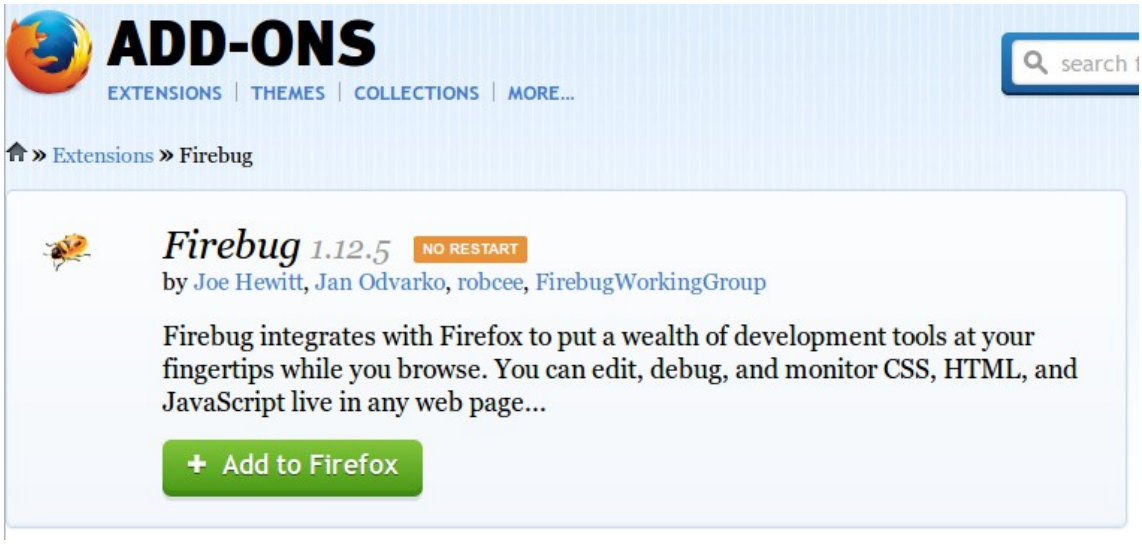
9. Tīmekļa lietotņu testēšanas rīki

9.1. Firebug spraudnis

Firebug ir Firefox pārlūkprogrammas spraudnis, kuru pamatā izmanto tīmekļa lietotņu izstrādei un atkļūdošanai: HTML, CSS un JavaScript koda izpētei un izpildes izsekošanai, resursu ielādēšanas pārraudzībai u.tml. Tomēr atsevišķos gadījumos Firebug var izmantot arī kā tīmekļa lietotņu testēšanas rīku.

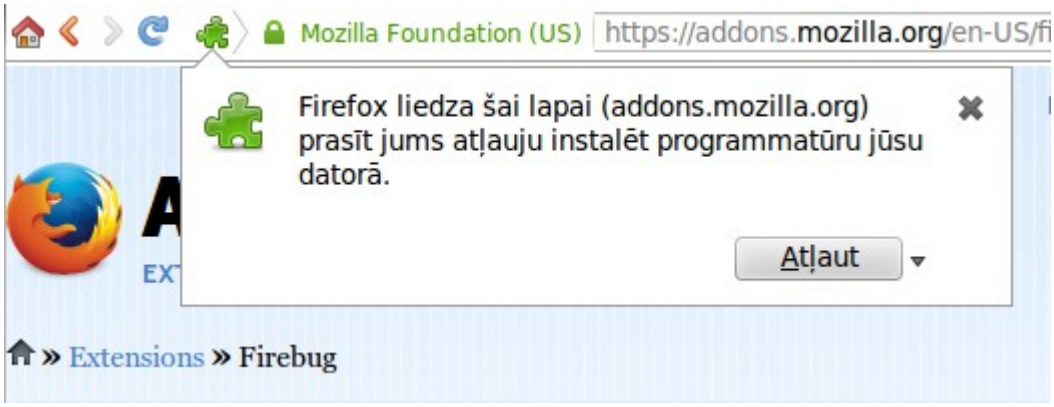
Firebug uzstāda, līdzīgi kā visus citus Firefox pārlūkprogrammas spraudņus.

Atver https://addons.mozilla.org/en-US/firefox/addon/firebug/ lapu un klikšķina uz pogas **Add to Firefox**:



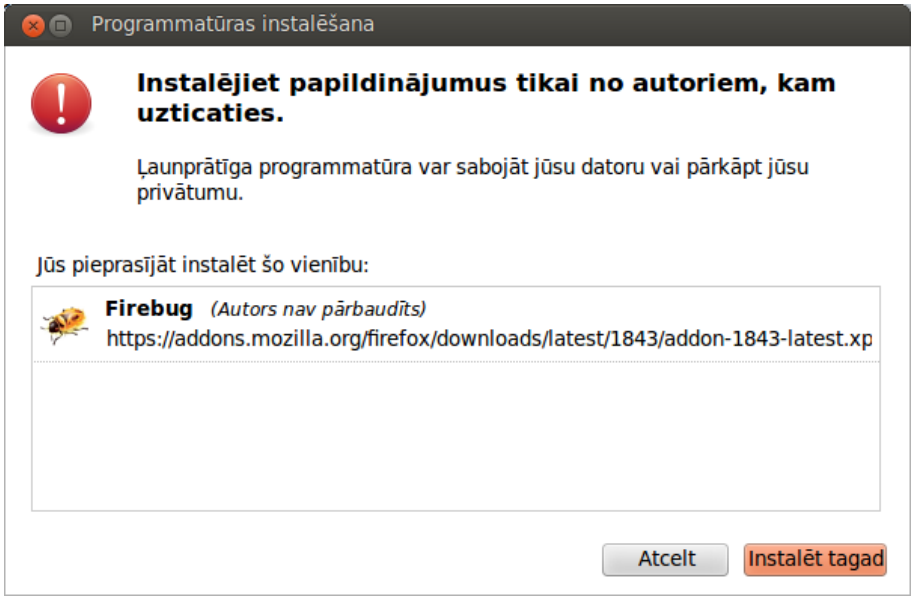
Attēls 9: Firebug spraudņa uzstādīšanas lapa

Spiež pogu **Atļaut** un uzgaida, kamēr notiek failu lejupielāde.



Attēls 10: Firebug spraudņa uzstādīšanas apstiprināšana

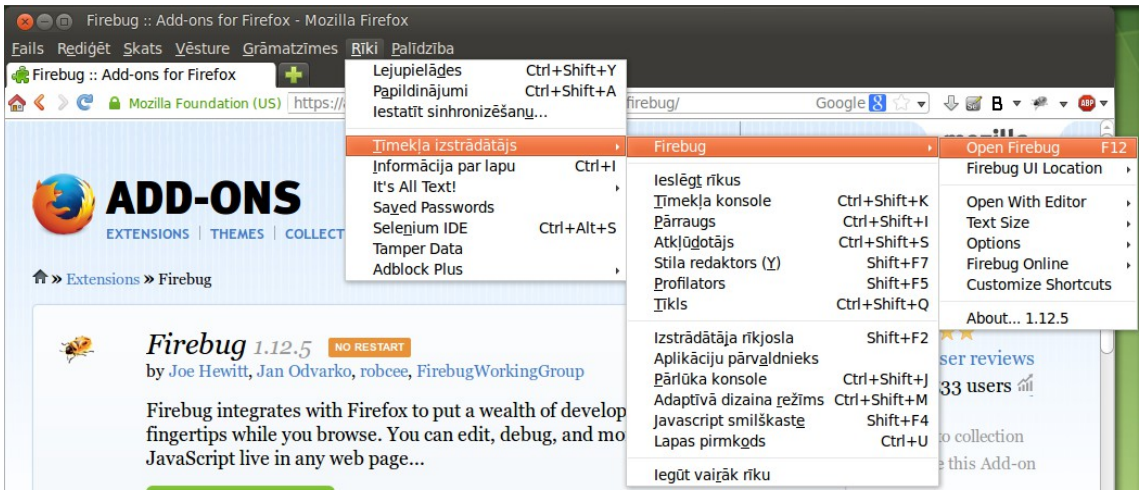
Kad faili ir lejuplādēti, uzstādīšanas logā uzgaida, kamēr parādās poga **Instalēt tagad** un spiež to.



Attēls 11: Firebug spraudņa uzstādīšanas apstiprināšana

Ja parādās uzaicinājums pārstartēt Firefox pārlūkprogrammu, izdara to.

Firebug spraudni palaiž, aktivizējot izvēlni: **Rīki— Tīmekļa izstrādātājs— Firebug— Open Firebug**, vai arī spiež vaboles pogu pārlūkprogrammas rīku joslā (kas parasti ir labajā augšējā stūrī).



Attēls 12: Firebug spraudņa aktivizēšana

9.2. Selenium IDE spraudnis

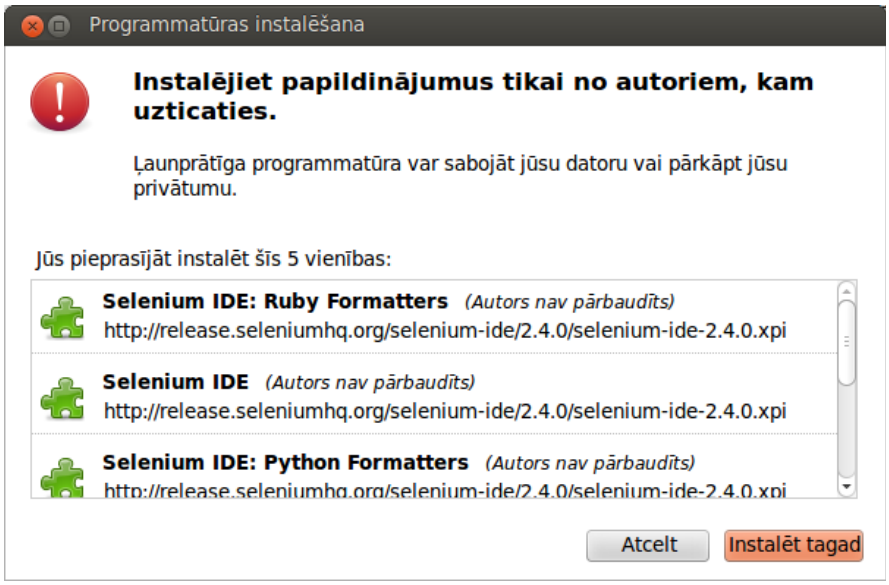
Selenium IDE ir Firefox pārlūkprogrammas spraudnis, ar kuru var ierakstīt un pēc tam atskaņot pārlūkprogrammā veicamās darbības.

Sarežģītākos gadījumos var izmantot Selenium Server rīku, kas darbojas programmēšanas vidē, piemēram Eclipse, un izsauc Firefox WebDriver API.

Lai uzstādītu Selenium IDE spraudni, ar Firefox pārlūkprogrammu atver vietni http://www.seleniumhq.org/download/ un klikšķina uz jaunākā laidiena hipersaiti, piemēram, 2.4.0

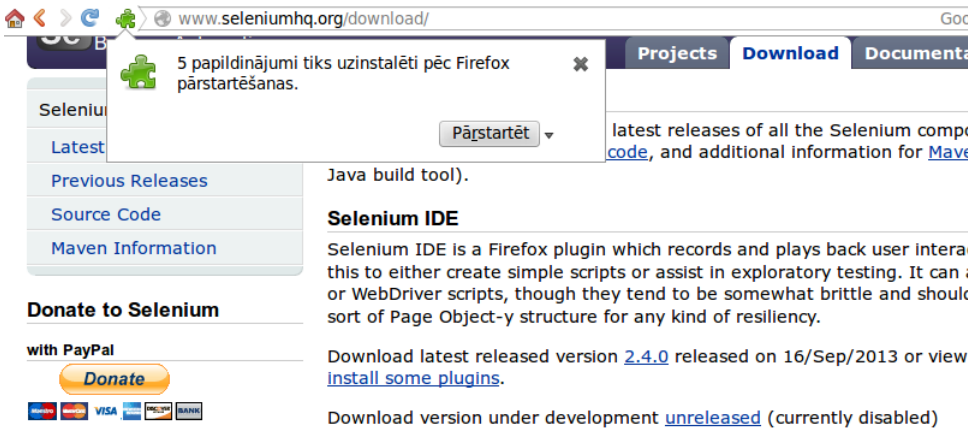
Attēls 13: Selenium IDE uzstādīšanas lapa

Spiež pogu **Atļaut** un uzgaida, kamēr lejuplādējas uzstādīšanas faili.



Attēls 14: Selenium IDE spraudņu uzstādīšanas apstiprināšana

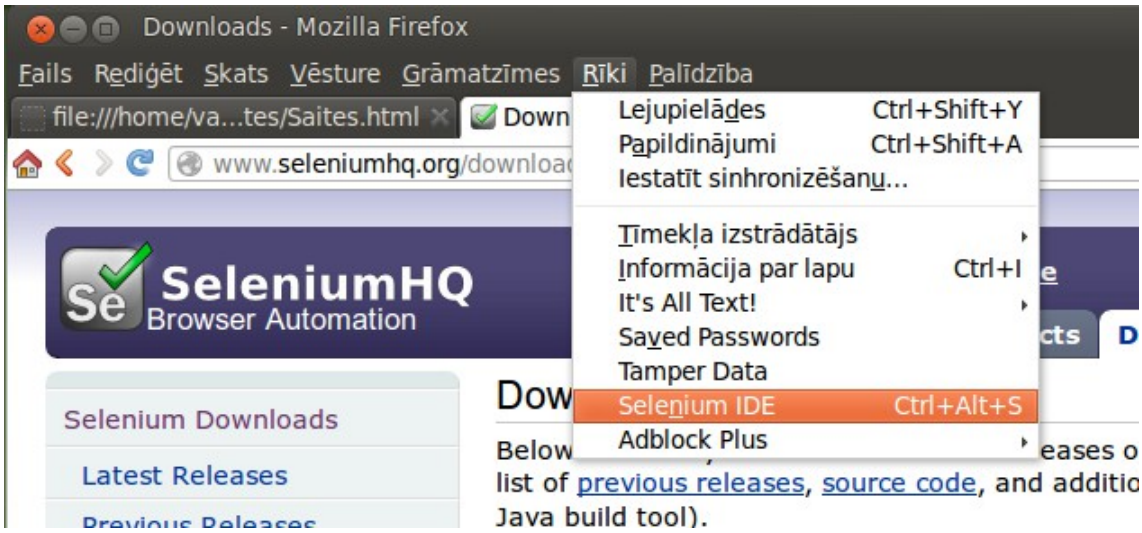
Uzgaida, kamēr spraudņu uzstādīšanas logā pogai parādās uzraksts **Instalēt tagad** un spiež to.



Attēls 15: Ziņojums pēc Selenium IDE spraudņu uzstādīšanas

Pēc veiksmīgas spraudņu uzstādīšanas parādās ziņojums, un pārstartē Firefox pārlūkprogrammu.

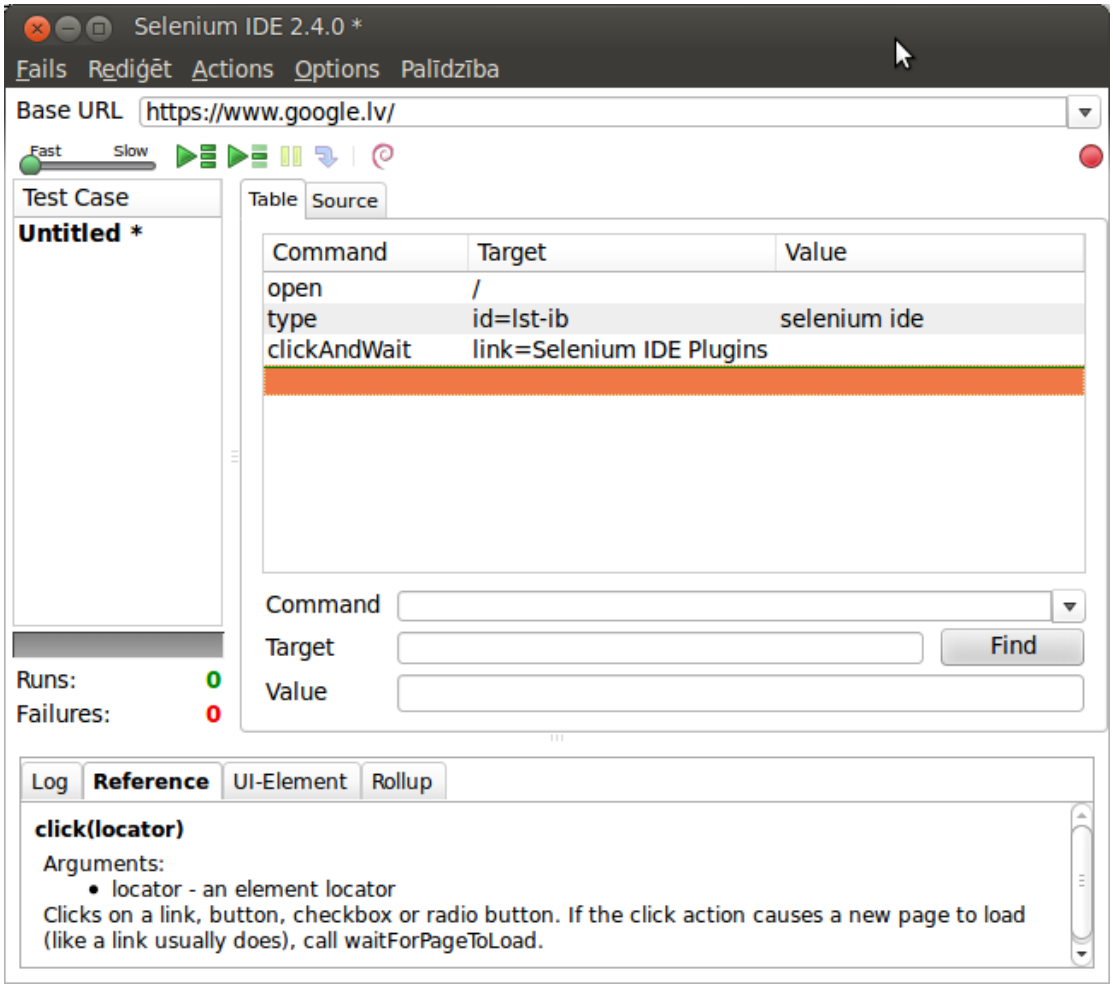
Pēc pārlūkprogrammas pārstartēšanas Selenium IDE palaiž, aktivizējot izvēlni: **Rīki — Selenium IDE**.



Attēls 16: Selenium IDE palaišana Firefox pārlūkprogrammā

Ievēro, ka palaižot spraudni, sarkanā poga labajā augšējā stūrī jau ir nospiesta un spraudnis ir gatavs sākt ierakstīt lietotāja darbības.

Logā **Base URL** var norādīt, relatīvi pret kādu adresi (vietrādi) darbības tiks ierakstītas. Tās darbības, kuras notiek, piemēram, https://www.google.lv/ adresē, tiks ierakstītas, vienkārši kā */*, jo relatīvi pret norādīto sākuma adresi tā ir vienkārši vietnes sākuma adrese.



Attēls 17: Darbību ierakstīšana ar Selenium IDE

Kad nepieciešamās darbības ir ierakstītas, spiež sarkano ierakstīšanas pogu, lai atslēgtu ierakstīšanu.

Ierakstītās darbības saglabā failā, spiežot **Ctrl+S** vai aktivizējot izvēlni **Fails— Save Test Case** un norāda faila nosaukumu un mapi.

Ierakstu atskaņo, spiežot zaļo trīsstūrveida pogu ar paskaidrojumu **Play Current Test Case.**

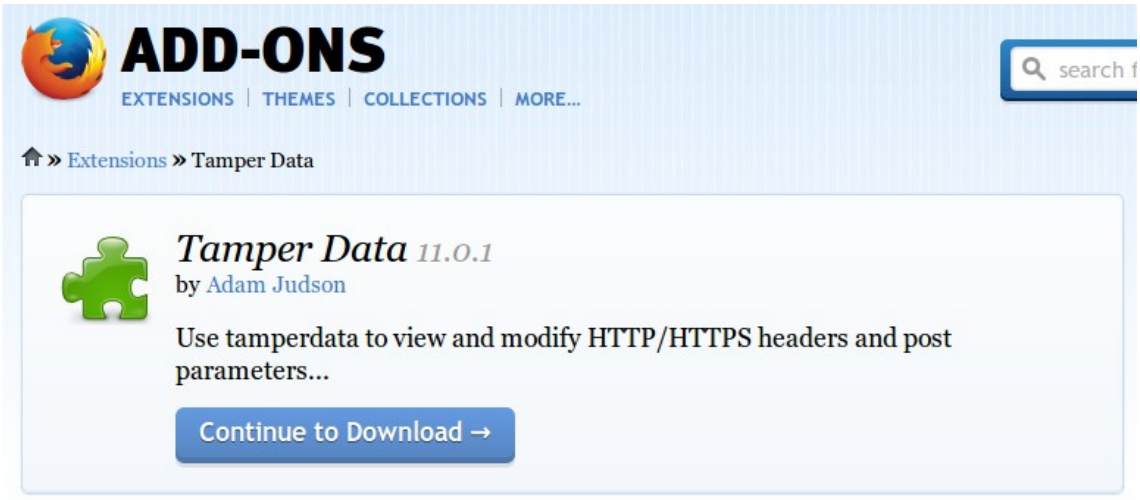
Vairākus testu piemērus (*test case*) var apvienot secīgā sarakstā un saglabāt testu komplektā (*test suite*).

9.3. Tamperdata spraudnis

Tamperdata ir Firefox pārlūkprogrammas spraudnis, ar kuru var ierakstīt un novērot no pārlūkprogrammas nosūtītos un no tīmekļa servera saņemtos datus. Nepieciešamības gadījumā, no pārlūkprogrammas nosūtītos datus var arī mainīt, ieskaitot gan HTTP hedera datus, gan GET vai POST vai citu HTTP pieprasījuma parametrus un to vērtību.

Lielākoties Tamperdata spraudni var izmantot, veicot negatīvos testus, lai nosūtītu tīmekļa serverim nepareizus datus, kādus "godīgi" lietojot pārlūkprogrammu lietotājs parasti nevar nosūtīt (piemēram, ja HTML formā lauks ir norādīts kā tikai lasāms, u.tml.).

Lai uzstādītu Tamperdata, ar Firefox pārlūkprogrammu atver https://addons.mozilla.org/en-US/firefox/addon/tamper-data/ un spiež pogu **Continue to Download**:

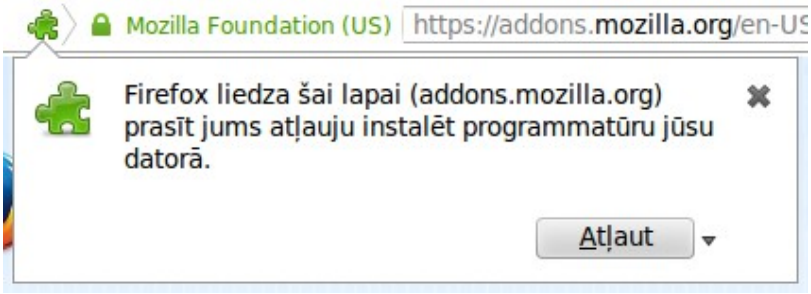


Attēls 18: Tamperdata spraudņa uzstādīšanas lapa

Atvērtajā licences noteikumu logā spiež pogu **Accept and Install**

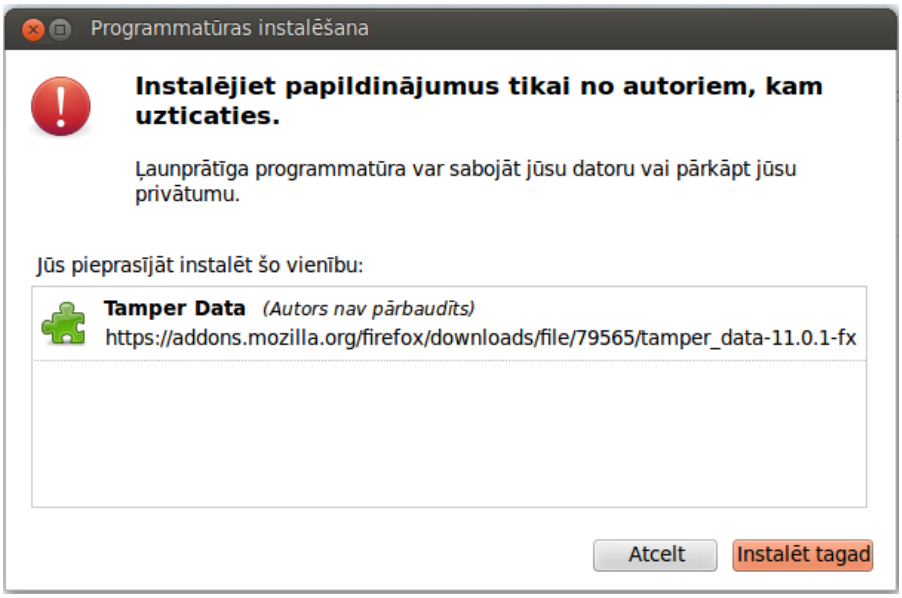
Attēls 19: Tamperdata spraudņa licences noteikumu logs

Spiež pogu **Atļaut**:



Attēls 20: Firefox spraudņa uzstādīšanas apstiprināšanas logs

Uzstādīšanas logā uzgaida, kamēr aktivizējas poga **Instalēt tagad** un spiež to.

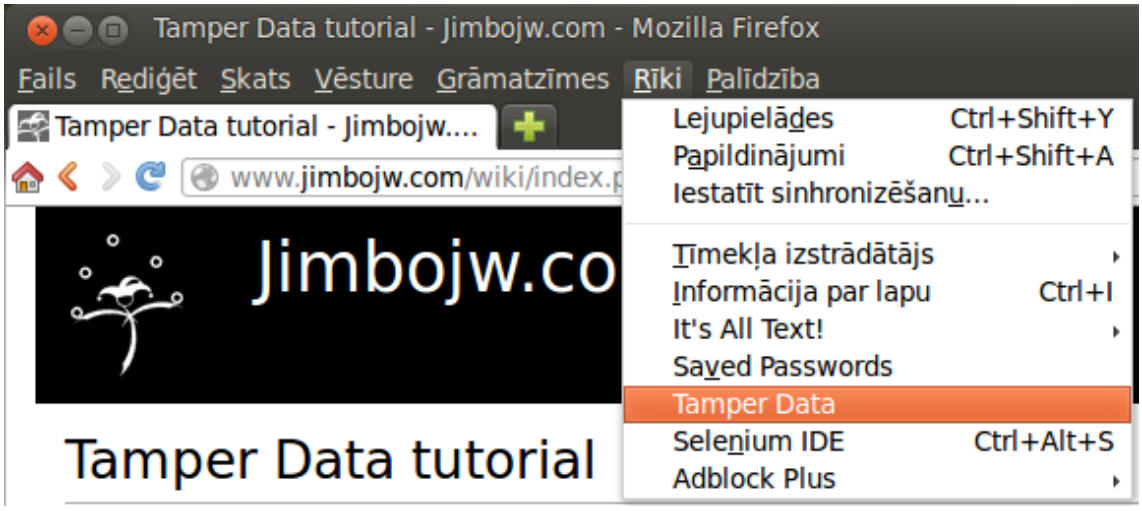


Attēls 21: Tamperdata spraudņa uzstādīšanas logs

Pēc spraudņa uzstādīšanas parādās logs par veiksmīgu spraudņa uzstādīšanu un nepieciešamību pārstartēt pārlūkprogrammu. Spiež pogu **Pārstartēt**.

Attēls 22: Tamperdata uzstādīšanas pabeigšanas paziņojuma logs

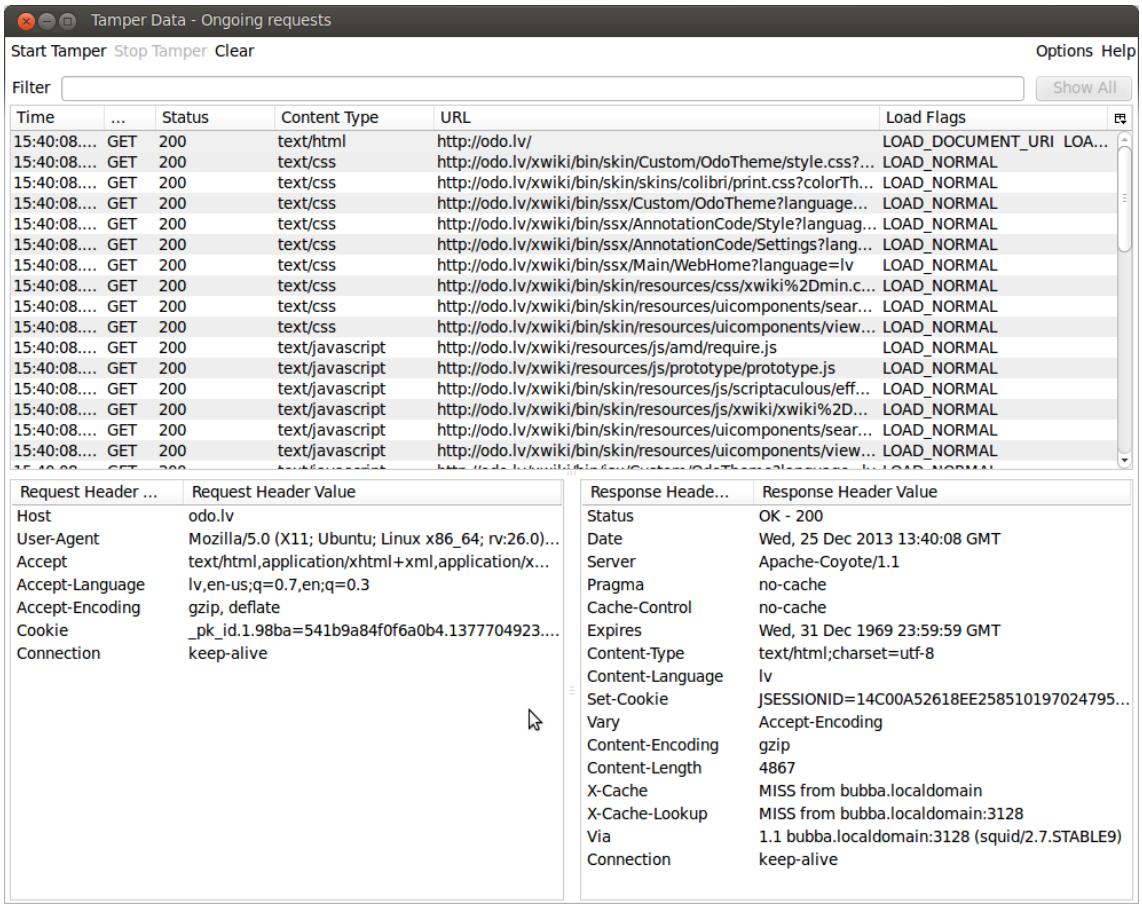
Tamperdata atver, aktivizējot izvēlni **Rīki— Tamper Data**



Attēls 23: Tamperdata spraudņa aktivizēšana

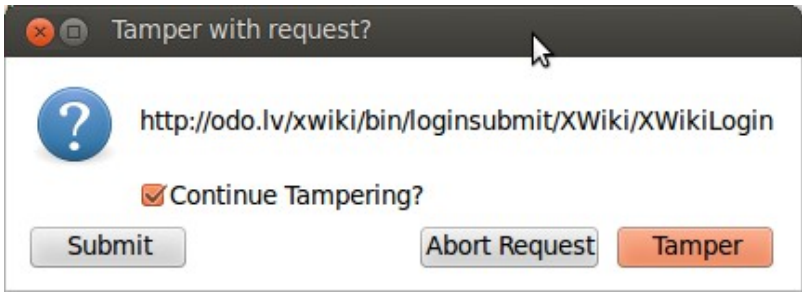
Aktivizējot Tamperdata spraudni, tas automātiski palaižas HTTP datu plūsmas ierakstīšanas režīmā un darbinot pārlūkprogrammu tajā tiek ierakstīti no

pārlūkprogrammas nosūtītie un no HTTP servera saņemtie dati.



Attēls 24: Tamperdata logs ar ierakstīto HTTP plūsmu

Nospiežot pogu **Start Tamper** loga kreisajā augšējā stūrī, katrā HTTP pieprasījumā lietotājam tiek piedāvāta iespēja, izmainīt no pārlūkprogrammas nosūtītos datus.

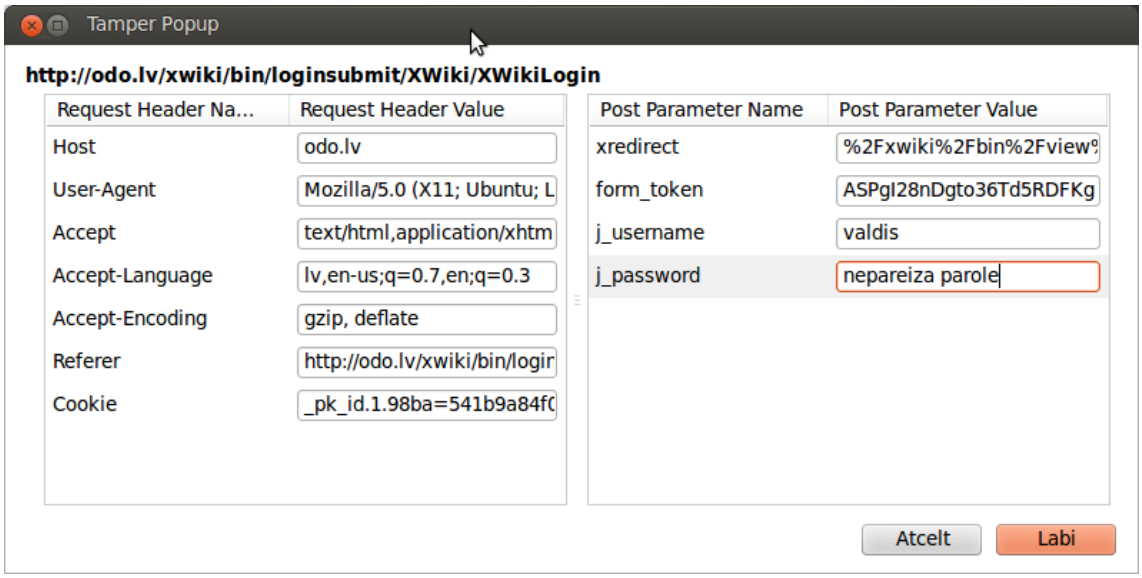


Attēls 25: Tamperdata datu mainīšanas apstiprināšanas logs

Jāņem vērā, ka sarežģītas tīmekļa lapas sastāv no vairākiem resursiem (attēliem, CSS un JavaScript failiem u.tml.), tāpēc "viena" tīmekļa lapa lietotājas skatījumā patiesībā var sastāvēt no daudziem (līdz pat daudziem desmitiem) pieprasījumiem un atbildēm.

Tālāk ir parādīts, kādus datus lietotāja pārlūkprogramma nosūta tīmekļa serverim, piemēram, lietotājam reģistrētojoties sistēmā.

Loga labajā pusē ir HTTP hederu nosaukumi (*Request Header Name*) un to vērtības (*Request Header Value*), ar kuriem pārlūkprogramma paziņo par sevi un iepriekšējo lapu, bet kreisajā pusē ir HTTP pieprasījuma (šajā gadījumā, Post pieprasījuma) parametri (*Post Parameter Name*) un to vērtības (*Post Parameter Value*).



Attēls 26: Parametru vērtību maiņa ar Tamperdata spraudni

Ievērojiet, ka visas gan HTTP hederu, gan Post parametru vērtības pirms nosūtīšanas tīmekļa serverim var mainīt. Kad vērtības ir nomainītas, spiež pogu **Labi**, lai nosūtītu datus tīmekļa serverim.

9.4. Apache benchmark

Viens no izplatītākajiem rīkiem tīmekļa servera veiktspējas testēšanai ir Apache benchmark, saīsināti **ab**.

Uzstādīšana Linux

Debian-veidīgā (piemēram, Ubuntu) Linux, ab uzstāda, ievadot komandu:

```
sudo apt-get install ab
```

un ievada lietotāja paroli.

Apache benchmark uzstādīšana Windows

Atver http://www.apachehaus.com/cgi-bin/download.plx un lejuplādē Apache paktotni, piemēram, **httpd-2.4.7-x86.zip**.

Atarhivē arhīvu. Spiež starta pogu un meklējamā komandā ievada **cmd** un spiež ievades (*Enter*) taustiņu.

Atvērtajā logā ievada norāda cd un atarhivētā arhīva vietu un ceļu uz izpildāmajiem failiem, piemēram,

```
cd Downloads\httpd-2.4.7-x86\Apache24\bin
```

Apache benchark lietošana

Vienkāršus, konkurējošus Get pieprasījumus ar Apache benchmark iegūst, ievadot, komandu:

```
ab -n 1000 -c 20 http://odo.lv/
```

kur:

-n 10000 norāda, ka tiks veikti pavisam 10000 pieprasījumi

-c 50 norāda, ka tiks veikti 50 vienlaicīgi/paralēli pieprasījumi

http://odo.lv/ ir pieprasījumu adrese

Ievadot šo komandu, tiek uzsākta pieprasījumu ģenerēšana.

Lai sekotu izpildes progresam, ab veic izdruku, ik pēc 100 izpildītiem pieprasījumiem.

Kad visi pieprasījumi ir pabeigti, tiek sniegts pārskats par rezultātiem.

```
valdis@vostro:~$ ab -n 1000 -c 20 http://odo.lv/
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking odo.lv (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests


Server Software:        Apache-Coyote/1.1
Server Hostname:        odo.lv
Server Port:            80

Document Path:          /
Document Length:        18886 bytes

Concurrency Level:      20
Time taken for tests:   70.196 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      19371000 bytes
HTML transferred:       18886000 bytes
Requests per second:    14.25 [#/sec] (mean)
Time per request:       1403.928 [ms] (mean)
Time per request:       70.196 [ms] (mean, across all concurrent requests)
Transfer rate:          269.49 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.3      0       5
Processing:   583 1398 706.1   1066    3085
Waiting:      549 1368 704.5   1037    3069
Total:        584 1399 706.1   1067    3086

Percentage of the requests served within a certain time (ms)
  50%   1067
  66%   1175
  75%   1400
  80%   2441
  90%   2663
```

```
  95%   2773
  98%   2865
  99%   2930
 100%   3086 (longest request)
```

Piemēram, šajā gadījumā, ir veikti vidēji 14,25 pieprasījumi sekundē, vidējais pieprasījuma apstrādes laiks ir bijis 1,403 sekundes un vidējais datu saņemšanas ātrums 299 kilobaiti sekundē. Jāpiezīmē, ka šajā testā šaurākā vieta ir nevis servera veiktspēja, bet gan testa datora pieslēguma ātrums internetam (10 Mb/s DSL pieslēgums), jo ir redzams, ka dati ir saņemti ar praktiski maksimālo DSL nodrošināto lejuplādes ātrumu.

Tāpēc intensīviem testiem ab ir jālaiž no datora, kurā ir liels pieslēguma ātrums, piemēram no cita datora, kas atrodas tajā pašā datu centrā un ir pievienots 100Mb/s vai pat 1Gb/s lokālajam tīklam.