Säkerhetstestning av Bittorrenttracker med fuzzingramverket SPIKE

JACOB NORDGREN
och ROBERT SVENSEN

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2010

Säkerhetstestning av Bittorrenttracker med fuzzingramverket SPIKE

J A C O B   N O R D G R E N
o c h   R O B E R T   S V E N S E N

Examensarbete i datalogi om 15 högskolepoäng
vid Programmet för datateknik
Kungliga Tekniska Högskolan år 2010
Handledare på CSC var Mads Dam
Examinator var Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/
nordgren_jacob_OCH_svensen_robert_K10068.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Sammanfattning

Fuzzing är en metod för att testa program genom att på ett automatiserat sätt generera testdata som sedan skickas till programmet i syftet att få det att krascha. Fuzzing utvecklades 1989 och har sedan dess fått fler och fler testområden från nätverksprotokoll till filformat. Många allvarliga säkerhetshål och buggar har upptäcks i alla möjliga program sedan dess med hjälp av fuzzing och är nu en del av varje mjukvarutestares (och hackares) verktygslåda. Bittorrent är en relativt ny teknik som används för att snabbt och effektivt dela filer och används av miljontals människor dagligen. Flera olika fuzzingstekniker och deras för- och nackdelar presenteras i rapporten. De flesta fuzzers skrivs idag med hjälp av ramverk, varav några kommer att presenteras här. Rapporten visar också hur en fuzzer för Bittorrenttrackers kan skrivas med hjälpa av fuzzing ramverket SPIKE, den testas också mot två stycken mycket välanvända trackers, Opentracker och Peertracker. Inga buggar eller säkerhetshål hittas i dessa program. Medvetna buggar introduceras därför i programmen och fuzzern körs mot dessa versioner av programmen i syfte att visa att fuzzern fyller sin funktion. Dessa buggar hittades av fuzzern och verifierar därmed fuzzerns funktionalitet.

Abstract

Security audit of Bittorrent trackers using SPIKE fuzzing framework

Fuzzing is a technique in software testing used for finding flaws in software by automatically generating and sending test data to a target program with the intent of making it crash. Fuzzing was developed in 1989 and have since grown to include all kinds of software and protocols ranging from network protocols to file formats. Many serious flaws have been discovered in all kinds of software using fuzzing and is of today a given tool in any tester or hackers arsenal. Many different kinds of fuzzing techniques have been developed and some of their pros and cons will be presented in this report. We will also show how a fuzzing module for Bittorrent trackers can be developed using the fuzzing framework called SPIKE. The module will also be used to test two know and well used trackers, Opentracker and Peertracker. No flaws were discovered in these programs and therefore we purposely introduced flaws in them to verify that the fuzzing module indeed is able to discover bugs. All the bugs introduced were found by the fuzzer.

Innehåll

1 Introduktion **1**
1.1 Förord . . . 1
1.2 Inledning . . . 1

2 Mjukvarutestning **3**
2.1 Testmetoder . . . 3
2.1.1 Whitebox testing . . . 3
2.1.2 Blackbox testing . . . 4
2.1.3 Greybox testing . . . 4
2.2 Testfall . . . 5
2.2.1 Ekvivalensklasser . . . 5
2.2.2 Gränsvärdesanalys . . . 5

3 Fuzzing **7**
3.1 Historia . . . 7
3.2 Olika typer av fuzzing . . . 7
3.2.1 Kommandorad . . . 8
3.2.2 Miljövariabel . . . 8
3.2.3 Filformat . . . 8
3.2.4 In-memory fuzzing . . . 8
3.2.5 Webbapplikationer . . . 9
3.2.6 Webbläsare . . . 9
3.2.7 Nätverksprotokoll . . . 9
3.3 Fuzzingtekniker . . . 9
3.3.1 Förgenererade . . . 10
3.3.2 Slumpmässig . . . 10
3.3.3 Manuell protokollmutation . . . 10
3.3.4 Mutation-bruteforce testing . . . 10
3.3.5 Automatisk protokollgenerering . . . 11
3.3.6 Fuzzdata . . . 11
3.4 Begräsningar . . . 12
3.4.1 Vad kan en fuzzer inte göra? . . . 12
3.4.2 Vad kan en fuzzer göra? . . . 12

4 Verktyg **15**
4.1 Ramverk 15
4.1.1 SPIKE 16

5 Konstruktion av fuzzermodulen **19**
5.1 Protokollet 19
5.1.1 Bittorrent 19
5.2 Testfall 21
5.3 Övervakning av programmet 21
5.4 Fuzzermodulen 22

6 Utvärdering **25**
6.1 Opentracker 25
6.1.1 Vad kan vi injicera? 25
6.2 Peertracker 27
6.2.1 Vad kan läggas till? 28

7 Resultat **31**
7.1 Diskussion 31

Litteraturförteckning **33**

Bilagor **35**

A Kod till fuzzermodulen. **35**

Kapitel 1

Introduktion

1.1 Förord

Arbetet med att bygga fuzzermodulen har i det stora hela skett i samarbete. Testerna mot peertracker genomfördes av Jacob medan testerna mot Opentracker genomfördes tille större del av Robert. Rapporten har till en större del skrivits av Jacob och någon särskild uppdelning har inte genomförts även om vissa delar nästintill uteslutande skrivits av en person. Typsättningen gjordes av Jacob i Latex efter Serafim Dahls mall för examensarbeten.

Vi vill också passa på att tacka Mads Dam som varit vår handledare och Johan Öhlin och Jonas Nordgren som tålmodigt korrekturläst och kommenterat rapporten.

1.2 Inledning

Få områden har upplevt en sådan snabb utveckling som datalogiområdet. Människor i allmänhet blir mer och mer beroende av datorprogram både i sina privatliv och arbete och använder datorprogram till allt fler ändamål. Detta har lett till att kraven på programutveckling också har förändrats, nya snabba utvecklingsmetoder och allt högre krav på tillförlitlighet och inte minst säkerhet.

Säkerhet inom programutveckling har gått från att vara något som hanterats lite i skymundan till att bli en av de viktigaste komponenterna i bra programvara. I och med att incitamenten för att hitta säkerhetshål i datorsystem har ökat då datorsystem hanterar mer pengar och information än någonsin tidigare har vi sett en lavinartad ökning av upptäckta säkerhetshål under de senaste åren. Allt detta leder till att det behövs nya metoder för att hitta säkerhetsrelaterade problem i program, en av de metoder som hamnat i fokus under de senaste åren är fuzzing.

Fuzzing bygger grovt sett på att man konstruerar ett program som medvetet försöker krascha ett annat program genom att skicka data till målprogrammet. Vi avser att med detta projekt beskriva bakgrunden till fuzzing, hur det fungerar, vad som kan uppnås och slutligen konstruera en fuzzer som ska kunna användas för att testa säkerheten i bittorrenttrackers. Fuzzern ska kunna ge en tillförlitlighet

mot vanliga välkända problem men kommer inte att ge någon heltäckande lösning och ska kunna användas mot alla typ av trackers oavsett implementation. Fuzzern kommer att konstrueras med hjälp av ett väletablerat ramverk. Fuzzern kommer att testas mot välkända och välanvända trackers och kommer att evalueras genom tester på originalmjukvaran. Samt om inga buggar hittas också testas mot modifierade versioner där säkerhetsproblem medvetet injicerats.

Vi kommer att börja med att beskriva de teoretiska och historiska grundprinciperna för fuzzing och sedan vilka metoder vi valt att använda och hur vi konstruerat fuzzern. Avslutningsvis kommer vi gå igenom hur vi testat fuzzern och vilka resultat vi kommit fram till.

Kapitel 2

Mjukvarutestning

Mjukvarutestning är den del av datalogin som handlar om testning av ett programs funktionalitet. Mjukvarutestning handlar i mångt och mycket om att göra det motsatta mot vad en utvecklare gör. En utvecklare försöker finna lösningar, en testare söker problem, vilket ger upphov till en del bakvänd logik. Ett lyckat testfall inom testning är ett testfall som gett upphov till ett fel. Forskning inom mjukvarutestning har genom åren gett upphov till ett antal olika metoder och tekniker av vilka vissa kommer presenteras här.

2.1 Testmetoder

Traditionellt inom mjukvarutestning finns det ett antal olika metoder som kan användas för olika testsituationer. Det första steget bör alltid vara att utgå ifrån vilken situation som är applicerbar på föremålet för testet. De möjliga situationerna kan delas in i tre olika huvudgrupper; whitebox, blackbox och greybox.

2.1.1 Whitebox testing

Whitebox testing kallas även logikdriven testning. Metoden bygger på att testaren i förväg har god tillgång till detaljerad information om testobjektet. Vad det praktiskt innebär kan variera från tillgång till programkod och direkt åtkomst till programmerarna som skrivit koden eller till detaljerad dokumentation. Grundtanken är hursomhelst att det finns tillgång till en stor kunskap om inte bara vad programmet gör utan också hur.[9]

Whitebox testing är en av de absolut vanligaste metoderna som används av utvecklarna själva ofta i form av en så kallad code review eller kodgranskning som är en formell process där utvecklare granskar varandras kod. En annan vanlig metod, på en något lägre nivå, är enhetstestning (unit-testing) där vanligtvis programmeraren själv skriver testfall för specifika kodstycken.

Mycket forskning har fokuserat på whitebox testing men praktiskt sett har det visat sig vara en ganska olämplig metod emellanåt, då det ligger i dess natur att

testaren följer programmets logik och programmerarens tankebanor alltför mycket. Ett konkret exempel på svagheter med metoden är när delar av källkoden till Microsofts Windows 2000 för några år sedan drabbades av en större läcka där källkod läckte ut på Internet. Läckan ledde däremot inte till mer än att en handfull säkerhetsluckor upptäcktes, varav inga fick några större konsekvenser.[10]

Inom modern forskning har whitebox testing oftast relaterats till problemet med programverifikation. Metoden kan ge väldigt genomgående testning av mycket större delar av koden än andra testmetoder eftersom tillgång till programmets logik också finns. Problemet ligger i att program ofta har väldigt komplicerade strukturer och behandlar komplicerad data och det blir därför svårt att använda kunskapen på ett bra sätt vilket ställer höga krav på testaren. Ett program som genomför fullständig whitebox testing vore målet för all mjukvarutestning.[5]

2.1.2 Blackbox testing

Blackbox testning är raka motsatsen mot whitebox. I blackboxsituationen är utgångspunkten väldigt lite kunskap om målet, ofta inte mer än kompilerade binära filer och kanske viss begränsad dokumentation. Blackbox är en relativt vanlig situation om tester utförs mot någon annans programvara. Det är oftast den utgångspunkt hackare utgår ifrån och är en situation där fuzzing ofta förekommer.

Fördelen är att det ger stor frihet för testaren att konstruera testfall utan fördomar från kännedom om logiken i programmet. Det kan resultera i att även obskyra testfall kommer med i testning. Nackdelen är att testningen gärna blir ganska ineffektiv och ställer höga krav på testarens intuition och erfarenhet. Historiskt sett har det ändå visat sig vara en mycket givande utgångspunkt som resulterat i upptäckten av mängder med buggar och säkerhetsluckor. Nackdelen är precis den samma som med fördelen med logikdriven. Det är enormt svårt för att inte säga omöjligt att konstruera genomgående testfall som testar hela programflödet. I listing 2.1 ges ett trivialt exempel på ett ett program som vore svårt att testa enbart som blackbox.[10]

Listing 2.1. Testas inte funktionen med argument lika med 17 kommer buggen aldrig att upptäckas.

```
int function(int i){
 if( i> 17)
         process(i);
 if(i < 17)
         process(1);
 if(i == 17)
         crashsystem(); //Pain
}
```

2.1.3 Greybox testing

Greybox är mellantinget mellan white respektive black box och går i princip ut på att man har tillgång till delar av dokumentation eller liknande. Greybox är vanligt vid testning av nätverksprotokoll eftersom god dokumentation ofta finns

tillgänglig om protokollet men inget om implementationen. Greybox har växt fram under senare år då mer standardiserade protokoll lett till att black och white inte beskrivit verkligheten tillräckligt väl.[10]

2.2 Testfall

Det första steget i att skapa en bra uppsättning testfall är att skaffa kunskap om vilken typ av indata som programmet behandlar samt vilka resultat programmet ger och utifrån denna information skapa testfall utifrån den informationen.

I en idealisk situation där det finns oändligt mycket resurser och tid för testning kan alla tänkbara testfall och kombinationer av dessa testas. I verkligheten är detta ofta ogenomförbart eftersom program oftast hanterar komplex indata vilket skulle resultera i en enorm mängd testfall. Ett bättre angreppssätt är att använda en vanlig teknik inom mjukvarutestning som kallas ekvivalensklasser och gränsvärdesanalys (boundary-value analysis).

2.2.1 Ekvivalensklasser

Ekvivalensklasser är en teknik för att koppla ihop olika testfall som mer eller mindre betyder samma sak vilket gör att endast ett av elementen i varje klass behöver testas. För varje indatakrav skapas sedan en korrekt ekvivalensklass såväl som en inkorrekt klass, namnen är ganska självförklarande men tanken är alltså att den korrekta klassen representerar korrekt indata enligt specifikationen och den inkorrekta felaktig indata som givetvis förväntas behandlas som sådan av programmet. [9]

Tabell 2.1. Ett simpelt exempel på ekvivalensklasser (notera att alla längre och kortare strängar också ingår i den inkorrekta klassen).

Indataspecifikation	Korrekt ekvivalensklass	Inkorrekt ekvivalensklass
ASCII strängar ska vara 5 tecken långa och börja med a.	aaaaa-azzzz	baaaaa - zzzzzz

Stora fördelen med välkonstruerade ekvivalensklasser är att de relevanta testfallen som bör testas inte blir ohanterligt många vilket är en mycket bra utgångspunkt för fuzzing.

2.2.2 Gränsvärdesanalys

Gränsvärdesanalys kan användas som en teknik för att hitta vilka fall inom ekvivalensklasserna som skall testas. Ekvivalensklasserna säger att endast en eller ett fåtal fall inom varje klass behöver testas och gränsvärdesanalysen svarar på vilka av

dessa som ger bäst avkastning.[9] Dessa är enligt metoden extremvärdena, alltså de värden som ligger på kanterna inom ekvivalensklassen i exemplet ovan vore aaaa, azzzz, baaaa, zzzzz samt ett antal mycket längre och kortare strängar vara lämpliga testfall. Givetvis är detta en mycket förenklad bild och i verkligheten är det sällan lika uppenbart som i exemplet och det finns ingen formell metod för vad som ska testas utan det krävs en stor mängd intuition och erfarenhet. Fuzzing har sin grund inom gränsvärdesanalys vilket kommer behandlas djupare i nästa kapitel.

Kapitel 3

Fuzzing

Fuzzing handlar om att på ett automatiserat sätt testa program efter buggar och säkerhetshål. Det finns ett antal olika metoder och typer av fuzzing av vilka vissa kommer att presenteras här. Fuzzing är ingen exakt vetenskap utan handlar mer om intuition och lite tur, paralleller har ibland dragits till fiske. Man är inte säker på att få napp men det är för de flesta ändå värt att försöka. Större delen av informationen i kapitlet kommer från boken ”Fuzzing: Brute Force Vulnerability Discovery”.[10]

3.1 Historia

Officiellt skapades fuzzing 1989 av professor Barton Miller vid University of Wisconsin. I realiteten har någon form av fuzzing existerat så länge det funnits programmerare då de flesta troligen vid något tillfälle testat att skicka data till ett program enkom för att försöka krascha det. Miller var den som först publicerat artiklar i ämnet, det var också han som först använde ordet fuzzing. Enligt myten härstammar fuzzing från störningar på en telefonledning under ett åskoväder vilket fick till följd att data som skickades över telefonledningen blev korrupt. Det var inget ovanligt i sig, däremot överraskades Miller av att datat kraschade diverse program på mottagarterminalen.[7]

Miller skrev 1990 ”Fuzz”, den första riktiga fuzzern. Miller använde från början fuzzing i en avancerad kurs i operativsystem där fuzzern i sig inte var mer än ett program som genererade slumpmässig indata som skickades till program via Unix-pipes.[1]

3.2 Olika typer av fuzzing

Fuzzing kommer i många olika former och skepnader men en övergripande indelning är lokalfuzzing och externfuzzing. Några exempel på användningsområden för lokalfuzzing är kommandoradsprogram, miljövariabler och filformat. Extern fuzzing

[1]Används i *nix system för att dirigera om strömmar.[15]

används bland annat för att fuzza webbapplikationer, webbläsare och nätverksprotokoll. Lokalfuzzing är ofta det mest tacksamma eftersom full åtkomst till själva processen oftast är möjlig. Detta möjliggör för testaren att koppla på ett externt program som kan kontrollera när och vad som händer om programmet kraschar. Detta är normalt inte möjligt vid extern fuzzing och då behövs andra metoder, en enkel metod kan vara att kontrollera att programmet svarar efter varje test och kontrollera den utdata som programmet skickar är korrekt.

3.2.1 Kommandorad

Vid kommandoradsfuzzing fuzzas de argument som ett program tar emot som indata när det startas och är vad fuzzing ursprungligen användes till. Det används mestadels i Unixmiljö och ofta mot setuid applikationer. Setuid används för att köra ett specifikt program med högre rättigheter än vad användaren har på systemet. Då kan programmet utnyttjas för att göra saker det inte var menat för, t.ex. att ändra på filer som användaren i normala fall inte har rätt att ändra på. iFuzz av Adam Greene är ett exempel på en kommandoradsfuzzer.

3.2.2 Miljövariabel

Ibland använder program miljövariabler för att få information om systemet det körs på. Ett program kan till exempel behöva veta var vissa systemfiler ligger, då används ofta miljövariabler för att lätt hitta dem. En miljövariabelfuzzer testar att ändra på olika miljövariabler för att se om programmet uppför sig annorlunda.

3.2.3 Filformat

En filformatsfuzzer genererar flera testfiler utifrån antingen en fungerande eller tom fil genom att till exempel flippa bitar, och öppnar dem sedan med målprogrammet. Sedan kontrolleras hur programmet hanterar dessa filer. Exempel på filformatsfuzzers är FileFuzz och SPIKEfile. FileFuzz är en fuzzer för Windows med ett grafiskt användargränssnitt. SPIKEfile bygger på ramverket SPIKE och är Linux baserat.

3.2.4 In-memory fuzzing

Om ett program är uppbyggt så att det tar lång tid att fuzza indata, t.ex. att man måste använda grafiska menyer för att skriva indata, så kan en kopia av hur processen ser ut i minnet sparas. Vid varje testfall öppnas kopian och skriver testdata direkt till minnet. Eftersom allt görs i minnet blir denna typ av fuzzing snabbare. En nackdel med det här sättet är att det kan ta lång tid att reproducera resultatet eller inte alls gå att göra utifrån programmet.

3.2.5 Webbapplikationer

I och med att många funktioner som tidigare bara fanns lokalt, t.ex. ordbehandling, fotoalbum och kalendrar nu har flyttat ut på Internet har intresset för fuzzing expanderat till att även innefatta applikationer på Internet. Alla dessa webbapplikationer går också att fuzza, men det kräver att fuzzern kan hantera HTTP. Fuzzing mot webbapplikationer riktar sig ofta mot säkerhetshål specifika för webben som SQL injects och XSS attacker.[2]

3.2.6 Webbläsare

Webbläsare har visat sig vara ett ypperligt lämpat mål för fuzzers. Anledning till det är många, dels hanterar webbläsaren väldigt komplicerade data som innefattar både kod, rörliga bilder och ljud. Dels är det ett intressant mål eftersom datat i sin natur kommer ifrån en extern källa vilket, potentiellt kan göra det möjligt att angripa systemet utan någon som helst åtkomst till den lokala miljön. En av de första webbläsarfuzzers som gjordes hette MangleMe som bland annat hittade en bugg i Internet Explorer. Buggen gjorde det möjligt att krascha webbläsaren enbart genom att sätta bredden och höjden på en bild till ett stort tal.[19] Några år efter MangleMe kom CSSDIE ut som till skillnad från MangleMe som testade html testade css, CSSDIE används än idag för att hitta felaktigheter i CSS protokoll. Framgångarna för bland annat MangleMe och CSSDIE fick fler att följa och det finns idag ett flertal webbläsar fuzzers. Fuzzers har visat sig såpass effektiva för webbläsaren att HD Moore (som är en av de absolut främsta experterna inom fuzzing) deklarerade juli 2006 som ”the month of browser bugs” genom att han varje dag under en månads tid publicerade en ny bugg.[8]

3.2.7 Nätverksprotokoll

Nätverksprotokollfuzzing omfattar allt som skickas över ett nätverk. Eftersom data skickas till en extern maskin och fuzzern måste vänta på svar så tar denna typ av fuzzing ofta längre tid än lokal. Ett sätt att snabba upp denna process är att, om programmet som fuzzas går att få tag på, köra programmet lokalt och använda loopbackadressen för input. Nätverksprotokollsfuzzing kommer senare i rapporten att användas mot bittorent protokollet.

3.3 Fuzzingtekniker

Den grövsta indelningen av tekniker vi kan göra är mutationsbaserad och genereringsbaserad. Mutationsbaserad bygger på att man ”muterar” förskapade testfall, gen-

[2]SQL Injection är ett vanligt säkerhetshål som utnyttjar svagheter i databas frågor med syfte att förvanska frågan, för att till exempel radera tabellerna eller komma åt information som inte borde vara tillgänglig. XSS eller cross-site scripting är ett annat säkerhetsproblem där man försöker utnyttja svagheter i en webbapplikation för att injicera skript i applikationen med syfte att andra besökare sedan exekverar skripten när de besöker sidan.[17][12]

erationsbaserade bygger istället på att man generar testfall från specifikationen av protokollet som är föremål för attacken.

3.3.1 Förgenererade

Förgenererade testfall är ofta en bra metod vid whitebox testing eller när man i förväg har en god uppfattning om hur och vad som skall testas. Metoden borgar också för goda möjligheter att återanvända testfallen till andra situationer. Den stora nackdelen är att det saknas en slumpmässig komponent och att det därför finns stora risker att många enkla fall som testaren inte tänkt på kommer att missas. En del fuzzerramverk, framförallt de lite äldre, använder sig av denna metod som då ofta inte är mer märkvärdig än att det finns en inbyggd lista med testfall som tidigare visat sig vara gynnsamma.

3.3.2 Slumpmässig

Slumpmässig är en metod där man som namnet antyder använder helt eller pseudoslumpmässig data. Fördelen är att man kan hitta mycket oförutsägbara sårbarheter. Nackdelen är att det givetvis tar en enormt lång tid eftersom väldigt många olika möjligheter måste testas för att få en bra spridning och därmed pålitlighet. Det blir också svårare att spåra exakt vad som gav upphov till problemet eftersom data kommer att förändras vid varje testomgång.

3.3.3 Manuell protokollmutation

I strikt mening är manuell protokollmutation inte en fuzzingteknik eftersom man då inte använder någon form av automation överhuvudtaget vilket som bekant är önskvärt vid fuzzing. Men detta är ändå den typ av fuzzing som alla programmerare troligen använder förr eller senare och den går helt enkelt ut på att en människa matar in testfall i programmet. Detta är givetvis ingen vidare effektiv metod men den har använts lyckosamt i vissa fall, till exempel fuzzing av webbläsare. En stor fördel är att man inte måste programmera in kontroller för att kontrollera när målprogrammet kraschar eller beter sig konstig utan personen kan använda sin intuition och tidigare erfarenhet till att upptäcka när det beter sig märkligt.

3.3.4 Mutation-bruteforce testing

Mutation-bruteforce testing går ut på att man först hittar en mängd med korrekt indata och sedan generar nya testfall utifrån dessa. Metoden används vanligtvis när man fuzzar filformat. Först skapas en korrekt fungerade fil och sedan flippas bitarna i den. Nackdelen är att detta givetvis tar enormt lång tid eftersom stora mängder indata, som målprogrammet inte kommer att behandla eller ens kunna läsa, kommer att genereras. Fördelen är att dessa fuzzers är ganska enkla att bygga och kan, om man har väldigt gott om tid, ge väldigt breda testfall.

3.3.5 Automatisk protokollgenerering

Automatisk protokollgenerering fungerar ungefär som mutation-bruteforce, skillnaden är att man med denna metod istället försöker bygga upp någon form av grammatik som sedan används för att generera nya testfall. Detta är den mest sofistikerade metoden som används av fuzzers och den ger ofta goda resultat. Nackdelen är att det är svårt att utveckla riktigt bra fuzzers. Ett exempel på verktyg som använder denna metod är SPIKE, som är det ramverk vi valt att använda för detta projekt. I SPIKE skrivs först ett skript som definierar den del av protokollet man vill fuzza och sedan generar SPIKE testfall utifrån detta.

3.3.6 Fuzzdata

En viktig fråga att ställa inför varje testsituation är vad som bör testas och hur. Vi har tidigare talat om ekvivalensklasser och gränsvärdesanalys men i verkligheten, framförallt i en blackboxsituation, finns ofta inte tillräckligt med information tillgängligt för att avgöra vilka klasserna kan tänkas vara. En mer generell metod är att nöja sig med att titta på vilken typ av data som programmet hanterar. Vanligtvis finns det två typer av indata ett program tar, strängar och heltal. Ren binär data är också tänkbar indata och är vanligt till exempel när målet är ett filformat. Men här nöjer vi oss med att behandla heltal och strängar som senare kan användas för att testa Bittorrenttrackern.

Heltal

Om idén med gränsvärdesanalys tas med i bilden kan ett antal vettiga testfall lätt identifieras eftersom en klassificerad gissning ofta kan göras om hur många bitar operativsystemet använder, vanligaste är som bekant 32 bitar eller 64 bitar. Och om en gissning inte kan göras kan säkerligen bägge två testas. I 32 bitars fallet identifieras fyra gränsvärden av intresse, det minimala värdet för en signed int -2,147,483,648, det maximala värdet, 2,147,483,647, samt 0 och 4,294,967,295 som är min- och maxvärdena för en unsigned int. Ett tänkbart mål för dessa värden är en funktion som kopierar data till minnet vilket skulle kunna resultera i en buffer overflow eller underrun.

Utöver min- och maxvärdena kan också diverse tal under och över dessa vara bra att testa då programmet kanske genomför operationer på talet innan minnesskrivningen. Programmet kanske kommunicerar med gamla funktioner som är kompilerade för ett system med färre bitar.

Strängar

Nackdelen med strängar kontra heltal är att de i princip kan vara hur långa som helst vilket gör det svårt att använda gränsvärdesanalys. Strängar används inte heller lika ofta i aritmetiska operationer vilket gör att en overflow troligen inte kommer att inträffa. Många strängar som är bra att testa är språkberoende.

Listing 3.1. Exempel på strängformateringsbugg i C.

```
int main(int argc, char *argv[]){
    printf(argv[0]);
}
```

Om strängen som skrivs ut kommer från indata, som i listing 3.1 , så kan strängformateringsfel introduceras om printf(indata) används istället för printf("%s", indata). Då tolkas indata som en formaterad sträng i första fallet och kan då vara ett säkerhetshål. Strängformateringfel är mycket allvarliga då de kan användas både till att läsa och skriva till godtyckliga minnesadresser[4]. Trots att strängformateringsfel kan förekomma i andra språk så är de så gott som exklusiva för C. Andra bra strängar att testa är gamla klassika problematiska strängar såsom väldigt långa strängar och strängar som försöker göra directory traversal såsom "//../../../../../../etc/passwd". Många av dessa strängar är välkända och finns inbyggda i de flesta ramverk, vilket vi kommer till senare, strängen ovan kommer från ramverket SPIKE.

3.4 Begräsningar

En fuzzer kan användas till att upptäcka flera olika typer av buggar men fuzzing är långt ifrån en perfekt lösning och det finns buggar som inte går att hitta eller är väldigt svåra att hitta med fuzzers.

3.4.1 Vad kan en fuzzer inte göra?

En fuzzer kan hitta många buggar/säkerhetshål i program, men en fuzzer kan inte försäkra att programmet som testas verkligen fungerar som det ska. En fuzzer kan vara ett verktyg för att kontrollera programmets funktionalitet men normalt sett är detta inte vad som menas med fuzzing.

Fuzzers hittar ofta inte komplexa buggar utan bara de enklare. En fuzzer skulle t.ex. kunna hitta en bugg i en server som gjorde att fuzzern fick administratorsrättigheter, men om inte buggen gjorde mer än så skulle fuzzern med stor sannolikhet inte märka det och därför inte rapportera buggen. Lindrigare minneskorruption behöver inte krascha programmet, t.ex. så skulle en godtycklig skrivning till minnet kunna hanteras av programmet utan att fuzzern märker det. Men det som bidrog till minneskorruptionen skulle kunna utnyttjas för att kringgå säkerhet i program vid en mer riktad attack mot programmet.[10]

3.4.2 Vad kan en fuzzer göra?

Fuzzers kan hitta buffer overflows, buffer underflows, strängformateringfel och många andra buggar. Moderna fuzzers har också visat sig framångsrika i att hitta mer subtila buggar, buggar som inte per definition kraschar program men ändå utgör allvarliga säkerhetshot, exempel på sådana buggar är SQL injection och XSS (crosssite scripting). För denna rapport har tydliga buggar varit det största målet, buggar

som påverkar program på ett sätt som gör att programmet kraschar eller returnerar saker det inte borde.

Kapitel 4

Verktyg

En fuzzer kan byggas helt och hållet från grunden och behöver inte vara svårare än några få rader kod.

Listing 4.1. Klassiskt exempel på en bufferoverflow fuzzer.[1].

```
perl -e 'print "A" x 5000'
```

Listing 4.1 är ett perl-skript som skriver ut 5000 A, det kan verka mycket trivialt men har visat sig vara ett obehagligt bra exempel på fuzzing. Realistiskt sett kommer däremot ett mer komplicerat program att krävas, en bra utgångspunkt är då att använda ett ramverk.

4.1 Ramverk

Fördelen med att använda ett ramverk är man får tillgång till mycket inbyggd information, det vill säga testfall som utvecklaren av ramverket ansett lämpliga. En annan stor fördel är att ramverken ger mycket färdigskriven funktionalitet som annars skulle ha behövts skrivas. Det kan till exempel vara funktioner för att skicka data över nätverk och datastrukturer för att representera olika protokoll. Vid test av ett trivialt program, till exempel ett terminalprogram med endast ett argument, kanske ett ramverk inte är nödvändigt. Men för något mera komplicerade protokoll och program är det i praktiken nästintill ett måste.

Det finns i dagsläget en uppsjö av olika ramverk tillgängliga både med öppen och stängd källkod. Skillnaden mellan ramverken kan ofta vara mycket små och många bygger på varandra, därför bör en stor del av valet baseras på vilket språk fuzzern ska skrivas i och vilken typ av mål den testar. Det mest kända och troligen ett av de mest använda ramverken heter SPIKE, som kommer särbehandlas något här eftersom det är ramverket vi valt för att konstruera trackerfuzzern.

Andra exempel på ramverk är Sully och Peach, bägge är skrivna i Python och påminner till viss del om SPIKE. Skillnaden ligger primärt i att Sulley och Peach är mer moderna och har mycket mer funktionalitet. Sulley (som är uppkallad efter monstret från filmen Monsters Inc.) innehåller utöver de klassiska funktionerna för

testfallsgenerering också moduler för att övervaka statusen på målet. Peach har gjort sig känt för att vara ett väldigt flexibelt ramverk där allt är uppdelat i mindre komponenter vilket möjliggör att kod ofta enkelt kan återanvändas. Utöver dessa finns det många fler såsom antiparser, dfuz och Autodaf (som sägs vara en modern version av SPIKE).[2]

4.1.1 SPIKE

Ett av de mest framgångsrika ramverken sett till antalet upptäckta buggar är SPIKE. Första versionen av SPIKE kom ut 2002 och skrevs av Dave Aitel. SPIKE kan på ett enkelt och smidigt sätt användas till att bryta ner komplicerade protokoll och testa dessa med inbyggda välkända testfall för mängder av olika applikationer. SPIKE konstruerades först och främst för att användas som en nätverksprotokoll fuzzer och innehåller färdigbyggda funktioner för att skicka data över både TCP och UDP.

Principiellt bygger SPIKE på att man bygger ett SPIKE-skript (.spk-fil) som matas in i en fuzzer. Denna fuzzer kan man skriva själv, men SPIKE innehåller redan ett antal, bland annat för att testa webbservrar och diverse olika RPC-protokoll.[1] Skriptet är skrivet i programspråket C och ska beskriva protokollet för applikationen. Skripten är avsedda att likna ett normalt testfall där själva variablerna som ska fuzzas är specificerade.

Listing 4.2. Exempel på delar av en pop3 fuzzer i SPIKE.

```
s_string_variable("USER");
s_string(" ");
s_string_variable("Administrator");
s_string("\r\n");
```

Koden i listing 4.2 betyder att administrator är en variabel som kommer att testas det vill säga den strängen kommer att bytas ut mot fuzzsträngar. Radbrytningar och USER är statiska strängar som aldrig kommer att ändras då dessa måste vara korrekta för att servern inte ska refusera datan på en gång. Statiska strängar ökar effektiviteten av fuzzingen eftersom det blir färre permutationer (annars skulle den statiska strängen behandlas som en fuzzsträng vilket resulterar i mängder med testfall som målprogrammet med största sannolikhet kommer att ignorera helt och hållet) nackdelen är att man riskerar att missa buggar.

Tanken är att SPIKE generar testfall utifrån detta och kontrollerar om programmet kraschar eller beter sig märkligt. Även om tanken är att protokollet ska representeras i ett skript finns det inget som hindrar att hårdkoda det in i programmet, vilket ur prestandasynpunkt kan vara en god idé eftersom man annars riskerar få en oönskad mängd filer öppna samtidigt.

En av de mest innovativa funktionerna med SPIKE är att protokollen kan representeras med hjälp av block. Främsta fördelen med block är att det blir lättare att konstruera testfall som har en större chans att accepteras. Detta eftersom många

[1] Remote Procedure Call, en teknik för att anropa funktioner i andra program, ofta på en annan dator i det lokala nätverket.[16]

protokoll ställer krav på att anropen ska specificera hur lång data som skickas (t.ex. kanske längden på en sträng måste specificeras). Skickas en längre sekvens än vad som anges i testfallet kan detta givetvis resultera att en bugg upptäcks, med det kan också resultera i att programmet helt enkelt ignorerar all data som överskrider gränsen. Lösningen är som sagt block, ett block är helt enkelt en metod som gör det möjligt att hålla koll på datan som skickas. Detta är mycket fördelaktigt när protokoll som använder flera lager testas. Ett exempel finns i listing 4.3.

Listing 4.3. Exempel på användandet av block i SPIKE.

```
s_block_start("exempel_block");
s_string_variable("USER");
s_string(" ");
s_string_variable("Administrator");
s_string("\\r\\n");
s_string("LENGTH");
s_blocksize_asciihex_variable("exempel_block");
s_block_end("exempel_block");
```

Notera i listing 4.3 användandet av s_block_size_ascihex_variable som kommer att skriva in längden på blocket under testning. Det finns i SPIKE ett trettiotal funktioner för att retunera blocksize beroende på vilken representation som är lämpligt för målsystemet.

SPIKE är framförallt byggt för att skicka och generera testfall och ”kraschkontrollen” är därför ganska primitiv. För de flesta fall kommer det krävas att utvecklaren själv bygger något eller använder ett externt program såsom en debugger. SPIKE har visat sig vara ett mycket effektivt ramverk och ett axplock av program den hittat buggar i är: RealServer, Server Message Block (protokoll från Microsoft som används för fildelning), Xeneo Web Server, ipSwitch (FTP server) med flera.[11]

Kapitel 5

Konstruktion av fuzzermodulen

5.1 Protokollet

Tillvägagångssättet för att hitta information om protokollet varierar med vilken situation testaren befinner sig i med avseende främst på blackbox, whitebox och greybox. Bittorrent är ett öppet och fritt protokoll men kunskap om implementationen kan inte tas för given. Att testa en bittorrenttracker är alltså en typisk greybox situation. Första steget bör då vara att sammanställa all information som finns tillgänglig och sedan komplettera med egna undersökningar.

5.1.1 Bittorrent

Bittorrent är en ganska komplex teknik som består av ett antal olika komponenter. Dessa är: en webbserver, en tracker och en klient som står för själva nedladdningen och kommunikationen med trackern och andra klienter. Det innovativa med Bittorrent är att det är en distribueringsteknik där varje fil delas upp i många små bitar som sedan sprids över alla klienter, dessa kan sedan dela med sig av bitarna till varandra. Detta gör att bandbredden som finns tillgänglig i nätverket av peers (direktanslutna klienter) ökar ju mer populär en torrent är, eftersom fler försöker ladda ner den och därmed förhoppningsvis också bidrar med sin egen bandbredd när de laddar upp till andra.

Vanligtvis när en användare försöker ladda ner en torrent så besöker denne först en vanlig webbserver och laddar där ner en fil med metadata, en torrentfil. I torrent filen finns information som beskriver hur bitarna som bygger upp filen eller mappen ser ut samt en URL eller adress till en eller flera trackers. Trackern kan beskrivas som den centrala servern i Bittorrent-nätverket och ansvarar för att dela ut information till alla anslutna klienter om vilka klienter som är anslutna och hur många delar av filen varje klient kan erbjuda. Denna information används sedan av respektive klient för att hitta andra klienter (som brukar kallas peers) från vilka klienten sedan kan ladda ner de bitar av filen som den behöver, målet för vår fuzzer är just denna data som den anslutna klienten skickar till trackern.

Trackeranropet går över HTTP och data skickas med GET metoden. Själva variablerna som är inblandade i standard bittorrent kan beskådas i tabell 5.1.[3]

Tabell 5.1. Tabell över fälten i ett tracker anrop.

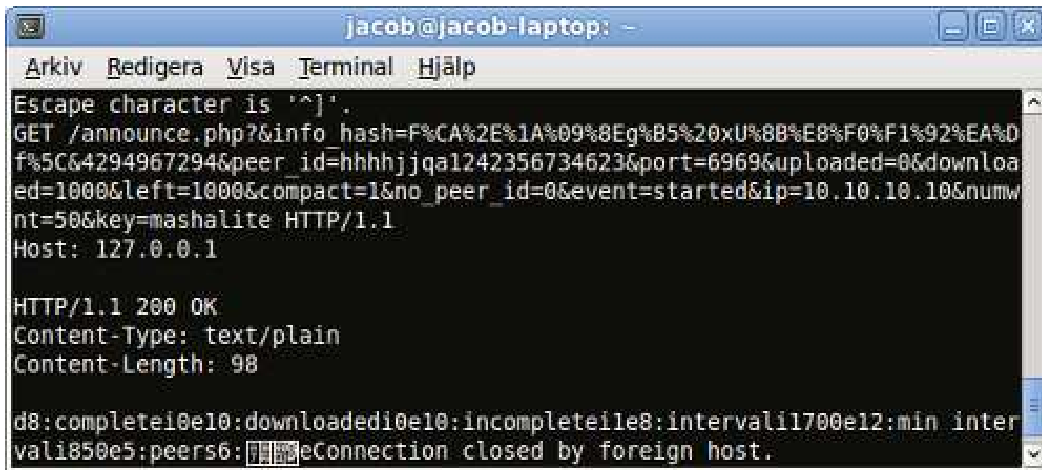
Variabel	Beskrivning
info_hash	En 20 byte stor SHA1 hash av innehållet efter ”info” taggen i torrent filen
peer_id	En slumpad sträng på 20 tecken genererad av klienten. Används som identifikation av klienten.
ip	Klientens IP adress eller domännamn.
port	Vilken port klienten lyssnar på.
uploaded	Hur mycket klienten har laddat upp till andra klienter.
downloaded	Hur mycket klienten har laddat ner från andra klienter.
left	Hur mycket klienten har kvar att laddat ner.
event	Kan ha värdena started, stopped och completed. Vilka sänds när klienten börjar ladda ner, slutar ladda ner och är färdig med torrenten.

Utöver dessa finns andra extra funktioner men då dessa oftast är implementationsspecifika och inte en del av standarden kommer dessa ej att behandlas närmare.

Eftersom alla data skickas över HTTP som är ett relativt simpelt och textbaserat protokoll kan det vara en god idé att titta på data som en normalt fungerande klient skickar till trackern. Om ett stängt protokoll varit mål för fuzzern hade detta troligen var det bästa angreppssättet men trots att vi i detta fall redan har stor kunskap om protokollet kan vi ändå nyttja en nätverkssniffer, t.ex. Wireshark, för att hitta ett korrekt exempel på data som skicks till trackern.

Härnäst bör svaren från trackern undersökas, i enlighet med specifikationen kommer svaren att vara ”bkodade” (bencoded) vilket innebär att strängar får formen [längd]:strängen, heltal kodas som i[talet]e. Listor kodas med ett l följt av antal element följt av elementen, l[antal element]:element, alla elementen är också de bkodade. Dictionaries kodas som ett ”d” följt av elementen där vart annat element är nyckel följt av dess värde delement. Element som kan finnas med i svaret är som följer: failure reason, warning message, interval, min interval, tracker id, complete, incomplete, peers.[3] De två första används endast om något gått fel oftast beror detta på en inkorrekt hashsumma. De övriga elementen är ren bkodad text förutom peer som kan vara formaterad på två olika sätt, antigen som en lista med dictionaries eller binärt. I det binära fallet är svaret en binär sträng som är en multipel av 6 där de första 4 bitarna representerar ip adressen och de sista 2 (för varje peer) är porten som används. [18] I figur 5.1 ser vi hur svaret från trackern ser ut. Anropet i figur 5.1 är skickat med telnet till en instans av bittorrent trackern Opentracker som körs

Figur 5.1. En korrekt fråga skickas via telnet mot Opentracker.



på loopback och utifrån svaret kan slutsatsen dras att Opentracker använder den binära modellen.

5.2 Testfall

SPIKE innehåller redan ett antal välkända testfall, upptill ca 6000 strängar finns inbyggt. Men beroende på vilka inställningar som används vid kompilering så kan många fler genereras. Det är också enkelt att lägga till egna strängar. Då SPIKE kom ut i början på 2000-talet och inte har utvecklats alltför mycket sedan dess, trots att hotbilden har förändrats, så är det ofta en bra ide att lägga till testfall för moderna säkerhetshål såsom SQL Injection. SPIKE har grundläggande tester för SQL injection men testar framförallt buffer overflow och bufferunderun vilket traditionellt har varit stora testområden för fuzzers. I den kompletta källkoden som finns tillgänglig via länken i bilaga A,kan dessa strängar studeras närmare.

5.3 Övervakning av programmet

Inom alla former av webbserverfuzzing och webbapplikationsfuzzzing är en av de absolut svåraste aspekterna att veta när man ”lyckats”, alltså när man fått servern/applikation att göra något det inte ska. Vid normal lokal fuzzing mot ett lokalt program kan man använda speciella program som debuggers och minnes monitorer. När man fuzzrar en extern server eller en serverapplikation har man inte den möjligheten utan man får förlita sig på andra metoder. Den enda informationen man normalt sett har att tillgång till är serverns svar och utifrån dessa får man skapa en metod för att hitta intressanta svar. Detta är ingen ideal lösning och kan inte ge upphov till ett exakt svar men fuzzing är inte heller någon exakt vetenskap. Problemet ligger

då i att filtrera svaren. Som exempel kan nämnas att om vår fuzzermodul skriver ut alla anrop och svar från servern till en fil blir den flera megabyte stor.

I fallet med Bittorrenttrackern är en bra utgångspunkt att ignorera svar när trackern (eller den underliggande servern) korrekt klassificerat frågan som inkorrekt. Eftersom all komunikation sker över http protokollet kan en bra utgångspunkt vara att använda några av de felkoder som inte är intressanta till exempel ”400 Invalid Request”, ”404 Not Found”.[6] En nästa filtrering kan vara att ignorera alla korrekta svar. Vi kan då utnyttja att första testfallet är ett korrekt uppsättning data vilket ger ett korrekt svar och kan därför hanteras ett särfall som alla andra svar sedan jämförs mot. En annan given filtering är tomma svar, dessa måste givetvis särskiljas från när eller om servern kraschar. En sista filtrering kan vara en funktion att ignorera svar som kommer gång på gång.

Detta är väldigt grova filtreringar men kommer ändå att minska mängden utdata betydligt och eftersom SPIKE saknar funktioner för debugging kommer en SPIKE modul alltid kräva ganska mycket manuell undersökning, alternativt extern programvara.

5.4 Fuzzermodulen

Relevanta delar av koden som inte presenterats här finns att beskåda i appendix. URL till den kompletta koden med tillhörande kompilerade program finns också i appendix.

Listing 5.1. Exempel initiering av en Spike.

```
struct spike * first_spike;
first_spike =new_spike();

 s_init_fuzzing();
```

Koden i listing 5.1 initierar en struct av typen spike som är den centrala delen av hela ramverket. Denna kod fyller en array med alla fuzzsträngar som ska användas. Själva automatiseringen av testet är inte svårare än att konstruera två loopar som går igenom all data vi avser testa, se listing 5.2.

Listing 5.2. Programloopen i en SPIKE modul.

```
while(!s_didlastvariable()){
    s_resetfuzzstring();

    //Looping over fuzzstrings.
    while(!s_didlastfuzzstring()){
        spike_clear();

        //Protocol description.

        if(spike_send_tcp(host, target_port) == 0){
            //Print error message
        }
        //Read server response.
        s_incrementfuzzstring();
    }
```

```
    s_incrementfuzzvariable();
}
```

I listing 5.2, s_didlastvariable() returnerar 1 om den sista variabeln har fuzzats klart och bryter då loopen. Annars så sätts arrayen med fuzzsträngar att börja om från början, med s_resetfuzzstring().

s_didlastfuzzstring() returnerar 1 om den sista fuzzsträngen har testats, och då anropas s_incrementfuzzvariable() som väljer ut nästa variabel att fuzza. Om det inte var den sista strängen så skapas en ny databuffer och protokollet läses in, sedan skickas den aktuella fuzzsträngen med spike_send_tcp() till målet. Om ingen anslutning kunde göras så kan ett felmeddelande skrivas ut, annars läses svaret från målet och kan visas på skärmen eller skrivas till fil. Det sista som händer i den inre loopen är att programmet går vidare till nästa fuzzsträng med metoden s_incrementfuzzstring().

Kodskelettet är allt som behövs för en modul och är taget direkt från modulen som vi skrivit. Det som inte finns representerat här är metoderna för att bearbeta svaren från servern och sedan slutligen skriva ut intressanta anrop och svar, samt metoderna som används för att beskriva själva protokollet. Närmare studier av koden kan göras i appendix för den intresserade läsaren.

Kapitel 6

Utvärdering

Meningen med en fuzzer är som sagt att hitta potentiella säkerhetshål och buggar i programmet den testar. Det enda rimliga sättet att utvärdera en fuzzer är därför att kontrollera om den hittar buggar. Tyvärr kan det vara mycket svårt (rent teoretiskt omöjligt) men för att verifiera att att fuzzermodulen i alla fall uppnått någonting kan en bra lösning vara ett injicera ett antal buggar i några trackers och sedan verifiera att den hittar dessa. Vi har valt att jobba med Opentracker och peertracker. Efter en körning av fuzzern mot dessa trackers finner vi inga problem, till fuzzerns försvar kan sägas att bägge programmen är välutvecklade till en såpass hög grad att stora organisationer använder dessa. Opentracker används bland annat på världens största tracker, the pirate bay samt denis.stalker.h3q.com och peertracker utvecklas bland annat av Bram Cohen som skapade Bittorrent.I dessa tester har version 1.229 av Opentracker, version 0.28 av libowfat och version 0.1.3 av Peertracker, vilket är de senaste versionerna vid tillfället för testningen.[13][14]

6.1 Opentracker

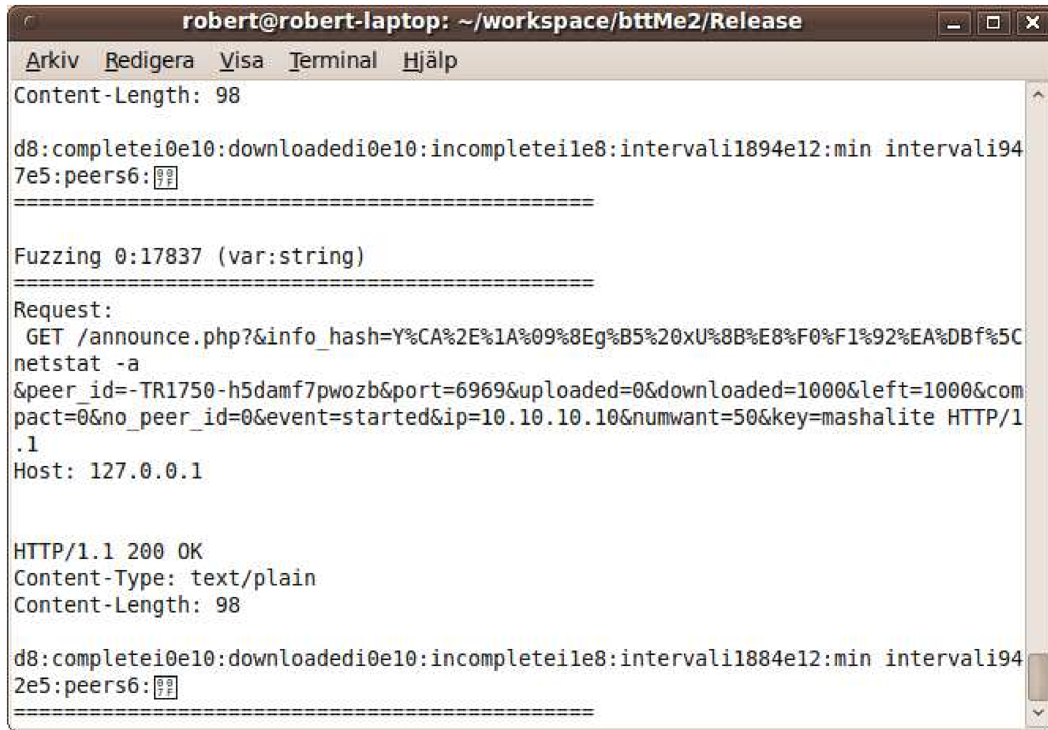
6.1.1 Vad kan vi injicera?

Opentracker är byggd i C och ett klassiskt säkerhetshål i C program är bufferoverflows låt oss därför se om vi kan introducera ett sådant.

Opentracker använder sig av libowfatbiblioteket för att ta emot data, för att kunna göra något mot Opentracker kan en bra utgångspunkt vara att modfiera libowfat. Ett första försök är att ta bort kontrollen på längden av bufferten som används vid inläsning av arrayer. I bilden nr 6.1 så ser vi att Opentracker inte skriver ut hela ip adressen på de peers som finns utan bara de första 2 byten, vilket skulle omöjliggöra trackerns funktionalitet.

Det som togs bort var if-satserna i array_get.c, från rad 20 till rad 23. Efter att ha undersökt fallen närmare verkar det inte vara någon specifik form av fuzzdata som får detta att hända, det enda som kan nämnas är att alla fallen sker vid fuzzing av den första variabeln (info_hash).

Figur 6.1. Fuzzing mot Opentracker med injicerad bugg.



Ett annat, större fel, som kan introduceras är om vi ändrar i ot_vector.c, från koden i listing 6.1 till koden i listing 6.2.

Listing 6.1. Rad 82 ur ot_vector.c.

```
memmove( match + member_size, match, ((uint8_t*)vector->data) +
    member_size * vector->size );
```

Listing 6.2. Modifierad rad 82 ur ot_vector.c.

```
memmove( match + member_size, match, ((uint8_t*)vector->data) + member_size
    * vector->size - 100  );
```

Vi har nu introducerat ett fel och vi kör fuzzern mot den modifierade versionen, utskriften från körningen kan ses i listing 6.3. Fuzzern stannar på variabeln 0 med strängen 2294 som är ”netstat -a”.

Listing 6.3. Utskriften från körningen av fuzzern.c.

```
Fuzzing 0:2294 (var:string)
Request:
 GET /announce.php?&info_hash=Y%CB%2E%1A%09%8Eg%B5%20xU%8B%E8%F0%F1%92%EA%DBf
     %5C
netstat -a
&peer_id=-TR1750-h5damf7pwozb&port=6969&uploaded=0&downloaded=1000&left=1000&
     compact=0&no_peer_id=0&event=started&ip=192.168.1.2&numwant=50&key=
     mashalite HTTP/1.1
Host: 127.0.0.1


***Server closed connection!
```

Nästa steg blir att verifiera att vi faktiskt hittade buggen vi sökte och ingen annan, vi kopplade därför också på en debugger, Valgrind, på Opentracker resultat som visas i listing 6.4, visar tydligt att ett fel inträffat på ”==4877== by 0x4081FC: vector_find_or_insert (ot_vector.c:82)” vilket är buggen vi introducerade.

Listing 6.4. Relevanta delar av utskriften från Valgrind.

```
==4877== Process terminating with default action of signal 11 (SIGSEGV)
==4877==  Access not within mapped region at address 0xF8E597B
==4877==    at 0x4C27138: memmove (mc_replace_strmem.c:613)
==4877==    by 0x4081FC: vector_find_or_insert (ot_vector.c:82)
==4877==    by 0x404115: add_peer_to_torrent_and_return_peers (trackerlogic.c
     :91)
==4877==    by 0x40B453: http_handle_announce (ot_http.c:510)
==4877==    by 0x40B5CD: http_handle_request (ot_http.c:573)
==4877==    by 0x402820: handle_read (opentracker.c:172)
==4877==    by 0x402BA1: server_mainloop (opentracker.c:261)
==4877==    by 0x403ECA: main (opentracker.c:618)
==4877==  If you believe this happened as a result of a stack
==4877==  overflow in your program's main thread (unlikely but
==4877==  possible), you can try to increase the size of the
==4877==  main thread stack using the --main-stacksize= flag.
==4877==  The main thread stack size used in this run was 8388608.
```

6.2 Peertracker

Peertracker är en relativt simpel tracker skriven i språket php. Den har inga funktioner mer än de allra nödvändigaste, inte minst därför är det föga förvånande att inga problem upptäcktes. Men för att verifiera att fuzzern gör vad den ska undersöker vi vad som skulle kunna finnas och försöker upptäckta dessa. Programmet består i princip av två filer announce.php samt tracker.mysql.php (namnet på denna fil varierar i enlighet med vilken databas trackern körs mot).

Genom att undersöka flödet i programmet vet vi att announce är indexfilen och därmed det ställe som först kommer att behandla datan. Utan att gå in på alltför mycket detaljer går den igenom all indata och genomför diverse kontroller på dessa. D fält som ska ha fix längd info_hash och peer_id kontrolleras om längden är 20. Är den inte det avslutas programmet omedelbart, detta förklarar varför vi fick relativt få svar vid körningar av fuzzern. Sedan kontrolleras att alla heltalsfält

faktiskt är numeriska och att ip adressen matchar avsändarens adress. Den enda variabel som inte kontrolleras i announce.php är event. Men en djupare undersök i tracker.mysql.php visar att variabeln endast används i ett switch sats vilket förefaller ganska ointressant. Tyvärr märker vi också att alla variabler går igenom PHP funktionen mysql_real_escape_string vilket är en funktion som blockerar eventuella farliga tecken. Vi kan alltså inte genomföra en SQL injection om vi inte lyckas överlista den funktionen.

6.2.1 Vad kan läggas till?

Det mest rimliga är att injicera en SQL injection. För att efterlikna en riktig situation väljer vi att lägga in vår bugg i en variabel som inte borde kunna innehålla några farliga tecken, nämligen porten. Efter en djupdykning ner i tracker.mysql.php hittas kodsstycket i listing 6.5.

Listing 6.5. Kodstycke ur tracker.mysql.php

```
"SET compact='" . self::$api->escape_sql(pack('Nn', ip2long($_GET['ip']),
    $_GET['port']))
```

Koden i listing 6.5 ligger i funktionen update_peer() som alltid anropas förutom när event inte är initierad. Tas escape_sql() bort så ska SQL kommandon kunna skickas till trackern utan att bli modifierade, så att koden istället blir som i listing 6.6.

Listing 6.6. Modifierad kod i tracker.mysql.php

```
"SET compact='" . /*self::$api->escape_sql(*/pack('Nn', ip2long($_GET['ip']),
    $_GET['port'])/*)*/ .
```

Peetracker kan hantera felaktiga frågor mot databasen och därför nöjer vi oss med att ta oss till ett felmedelande som antyder att något skett med databasen.

Listing 6.7. Första testfallet

```
Fuzzing 0:0 (var:string)
================================================
Request:
 GET /tracker/announce.php?&info_hash=Y%CA%2E%1A%09%8Eg%B5%20xU%8B%E8%F0%F1
    %92%EA%DBf%5C&peer_id=-TR1750-h5damf7pwozb&port=6969&uploaded=0&
    downloaded=1000&left=1000&compact=0&no_peer_id=0&event=started&ip
    =192.168.1.2&numwant=50&key=mashalite HTTP/1.1
Host: 127.0.0.1


HTTP/1.1 200 OK
Date: Wed, 21 Apr 2010 03:02:38 GMT
Server: Apache/2.2.11 (Win32) PHP/5.3.0
X-Powered-By: PHP/5.3.0
Content-Length: 112
Content-Type: text/html

d8:intervali1800e12:min intervali900e5:peersld2:ip11:192.168.1.27:peer id20:-
    TR1750-h5damf7pwozb4:porti65535eeee
================================================
```

Vid första körning ger första testfallet i listing 6.7 ett korrekt svar vilket är precis vad som förväntats eftersom fuzzern alltid skickar korrekt data första iterationen för varje variabel. Samtliga testfall för första variabeln det vill säga info_hash ger alltid felaktigt svar eftersom strängarna inte blir 20 tecken långa och därför blir ignorerade Andra variabeln det vill säga peer_id ger precis samma resultat av samma anledning vilket är väntat eftersom det bara innebär att programmet fungerar.

Listing 6.8. Andra testfallet på port fältet

```
Fuzzing 2:1 (var:string)
================================================
Request:
 GET /tracker/announce.php?&info_hash=Y%CA%2E%1A%09%8Eg%B5%20xU%8B%E8%F0%F1
     %92%EA%DBf%5C&peer_id=-TR1750-h5damf7pwozb&port=/.:/
     AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
     \-
     AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

HTTP/1.1 200 OK
Date: Wed, 21 Apr 2010 03:03:53 GMT
Server: Apache/2.2.11 (Win32) PHP/5.3.0
X-Powered-By: PHP/5.3.0
Content-Length: 54
Content-Type: text/html

d14:failure reason32:client listening port is invalide
================================================
```

Det andra testfallet (i listing 6.8) på porten ger däremot ett mer intressant resultat, vi har tidigare tagit bort skyddet mot SQL injection och detta felmedelande måste komma från en annan kontrollfunktion. Därför får vi modifiera ytterligare en rad kod, den rad som kontrollerar att porten är ett heltal.

Listing 6.9. Testfall nummer 11 på port fältet utan SQL injection skydd och heltals koll.

```
Fuzzing 2:11 (var:string)
================================================
Request:
 GET /tracker/announce.php?&info_hash=Y%CA%2E%1A%09%8Eg%B5%20xU%8B%E8%F0%F1
     %92%EA%DBf%5C&peer_id=-TR1750-h5damf7pwozb&port
     =../../../../../../../../../../../../etc/shadow%00&uploaded=0&downloaded
     =1000&left=1000&compact=0&no_peer_id=0&event=started&ip=192.168.1.2&
     numwant=50&key=mashalite HTTP/1.1
Host: 127.0.0.1


HTTP/1.1 200 OK
Date: Wed, 21 Apr 2010 02:39:46 GMT
Server: Apache/2.2.11 (Win32) PHP/5.3.0
X-Powered-By: PHP/5.3.0
Content-Length: 159
Content-Type: text/html

d8:intervali1800e12:min intervali900e5:peersld2:ip11:192.168.1.27:peer id20:-
    TR1750-h5damf7pwozb4:porti6969eeeed14:failure reason26:failed to update
    peer datae
================================================
```

Vid en andra körning liknar alla anrop och svar som fuzzern skriver ut, det som visas listing 6.9 och kommer från fuzzing av porten. Vilket skulle ge oss en antydan om att det är något speciellt med det fältet. Av felmedelanden kan vi också dra en slutsats om vad som hänt, nämligen att något inte lyckats uppdatera peer listan. Vilket betyder ett problem med databasfrågan eftersom programmet använder en databas. Att faktiskt inse att det finns möjligheter till en SQL injection hade varit mycket svårare om vi inte visste det från början givetvis. Men ofta kan fuzzers inte ge mer än en antydan om var fortsatta undersökningar bör genomföras. Ett välskrivet program som peertracker ger ofta inte mycket mer information än så här eftersom felhantering finns inbyggd. Ett test mot en sämre trackermjukvara skulle kunna ge ett mer explicit felmeddelande kanske till och med från själva databashanteraren. Men vi kan ändå bedömma testet som lyckat eftersom fuzzern klart och tydligt pekar på ett problem med fältet där vi introducerade buggen.

Kapitel 7

Resultat

Resultatet är en fungerande fuzzer som bevisats klara av att hitta vissa säkerhetshål. Fuzzing är ingen exakt vetenskap lika lite är fuzzermodulen det. Men vi har ändå i denna rapport visat hur en fullt funktionsduglig fuzzer kan konstrueras och visat att den klarar av att hitta medvetna säkerhetshål.

7.1 Diskussion

Fuzzing är en teknik som funnits i ett antal år och har i praktiken visat sig vara en mycket effektivt metod för att hitta säkerhetshål. Fuzzing har till väldigt stor del använts av säkerhetsforskare (och hackare) och sedan några år tillbaka har även Microsoft inkluderat fuzzing som en del i SDL (Security development lifecycle). SDL används både av Microsoft själva och flera andra. Fördelarna med att använda fuzzing är många och även om det finns svagheter vägs dessa upp av fördelarna. Säkerhet har under senare år även visat sig ha stora ekonomiska konsekvenser, enligt FBI stod enbart virus för kostnader på 16 miljoner USD. Med det i baktanke framstår fuzzing som en billig metod. Fuzzing är med moderna ramverk inte heller någon särskilt komplicerad teknik att använda.

En rekommendation till en blivande fuzz testare är däremot att använda ett modernare ramverk. Tidens tand har gått hårt åt SPIKE och det finns egentligen inga större skäl att använda det idag utöver historiska skäl. Vi valde tidigt att jobba med SPIKE eftersom det är svårt att använda, och det finns endast bristfällig dokumentation vilket tvingar användaren att sätta sig in i koden. Men i verkligheten är det bättre att nyttja färdigutvecklade verktyg med bättre dokumentation och exempel, istället för att brottas med själva ramverket kan man då istället fokusera på målet man avser testa. Vi valde medvetet ett ganska enkelt mål för vår fuzzer och därför klarade SPIKE uppgiften förhållandevis bra men ett mer avancerat mål hade ställt andra krav och ett annat ramverk borde då använts.

Eftersom hackare också har tillgång till fuzzing så kan det vara bra för företag att själva fuzza sina produkter först så att det kan hitta, i alla fall de mest uppenbara buggarna, och rätta till dem så att ekonomiska skador kan minimeras. Då fler och fler

tjänster flyttas ut på Internet och den mängden information datorsystem hanterar ökar, betyder att säkerheten måste tas på allvar. Fuzzing är på egen hand i dagsläget ingen lösning på säkerhetsproblemen men vi har med denna rapport försökt visa att det kan vara ett bra verktyg.

Litteraturförteckning

[1] Dave Aitel. An introduction to spike, the fuzzer creation kit. http://www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt, 3/5 2010.

[2] Pedram Amini and Aaron Portnoy. Blackhat us. In *Fuzzing Sucks! Introducing Sulley Fuzzing Framework*, 2007.

[3] Bram Cohen. The bittorrent protocol specification. http://www.bittorrent.org/beps/bep_0003.html, 29/4 2010.

[4] Jon Erickson. *Hacking: The Art of Exploitation.* No Starch Press, San Franscisco, California, 2003.

[5] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. Technical report, Microsoft,UC Berkeley.

[6] Network Working Group. Rfc 2616. http://tools.ietf.org/html/rfc2616, 3/5 2010.

[7] Barton Miller. Fuzz testing of application reliability. http://pages.cs.wisc.edu/\~bart/fuzz/, 9/3 2010.

[8] HD. Moore. Month of browser bugs. http://blog.metasploit.com/2006/07/month-of-browser-bugs.html, 29/4 2010.

[9] Glenford Myers, Tom Badgett, Todd Thomas, and Corey Sandler. *The Art of Software Testing, second edition.* John Wiley & Sons, Inc., Hoboken, New Jersey, 2004.

[10] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley, Upper Saddle River, New Jersey, 2007.

[11] Ilja van Sprundel. Chaos conference. In *Fuzzing*, 2005.

[12] Wikipedia. Cross-site scripting. http://en.wikipedia.org/wiki/Cross-site_scripting, 18/6 2010.

[13] Wikipedia. Opentracker. http://en.wikipedia.org/wiki/Opentracker, 3/5 2010.

[14] Wikipedia. Peetracker. http://en.wikipedia.org/wiki/PeerTracker, 3/5 2010.

[15] Wikipedia. Pipeline (unix). http://en.wikipedia.org/wiki/Pipeline_(Unix), 3/5 2010.

[16] Wikipedia. Remote procedure call. http://en.wikipedia.org/wiki/Remote_procedure_call, 3/5 2010.

[17] Wikipedia. Sql injection. http://en.wikipedia.org/wiki/SQL_injection, 18/6 2010.

[18] wiki.theory.org. Bittorrent protocol specification v1.0. http://wiki.theory.org/BitTorrentSpecification, 29/4 2010.

[19] Mike Zusman. Penetration testing and vulnerability analysis. http://cryptocity.squarespace.com/fuzzing/, 9/3 2010.

Bilaga A

Kod till fuzzermodulen.

Hela koden samt kompilerad version kan finnas på:

http://www.csc.kth.se/~jacobnor/dkand/
http://www.csc.kth.se/~rsvensen/dkand/

Koden nedan går inte att kompilera och är endast till för att illustrera programmet.

Listing A.1. De mest intressanta delarna ur fuzzer koden

```
int main(int args, char *argv[]){

        if(args>0){
                parseArgs(args,argv);
        }

        int current_fuzzvar = 0;
        int current_fuzzstring = 0;

        int retval;

        struct spike * first_spike;
        first_spike =new_spike();

        s_init_fuzzing();
        //adds evil!
        jakesFuzzstrings();

        if(first_spike == NULL){
                printf(stderr,"Couldnt not build spike, fatal error!");
                exit(-1);
        }

        setspike(first_spike);

        //Looping over fuzz variables.
        s_resetfuzzvariable();
        while(!s_didlastvariable()){
             s_resetfuzzstring();

             //Looping over fuzzstrings.
                while(!s_didlastfuzzstring()){
```

```
                    spike_clear();

                    /** HÄR SKA SCRIPTET LIGGA **/
                    testData();
                    //Retunerar 1 vid framgång!
                    if(spike_send_tcp(host, target_port) == 0){
}else{
/**
* Mer kod här
**/


                    //Sparar svaret från servern
                    memset(responseBuffer, 0x00, sizeof(responseBuffer));
                    retval=read(first_spike->fd,responseBuffer,2500);
                    if ((retval==-1 || retval==0) ){

                            printf("\nFuzzing %d:%d (var:string)\n",
                                current_fuzzvar,current_fuzzstring);
                            printRequest();
                            fprintf(stderr,"***Server closed connection!\
                                n");
                            exit(0);
                            break;
                    }
                    /**
                     * Sållar bort responses vi inte är intersserade av.
                     */
                    else if(!strstr(responseBuffer, "400 Invalid Request"
                        ) &&
                                    !strstr(responseBuffer, "404 Not
                                        Found") &&
                                    !strstr(responseBuffer, "414 Request-
                                        URI")
                                    && !filterBlankResponses()){

                            //Plockar ut ett korrekt response
                            if(!first){

                                    char *tmp = strstr(responseBuffer,"\r
                                        \n\r\n");
                                    strcpy(correctResp,tmp);
                                    //first = true
                                    first = 1;
                            }
                            printf("\nFuzzing %d:%d (var:string)\n",
                                current_fuzzvar,current_fuzzstring);
                            printf("
                                ==================================================\
                                n");
                            printRequest();
                            printf("%s\n", responseBuffer);
                            printf("
                                ==================================================\
                                n");
                    }

                spike_close_tcp();
                    }
                    current_fuzzstring++;
                        s_incrementfuzzstring();


            }
```

```
                current_fuzzstring = 0;
                current_fuzzvar++;
                s_incrementfuzzvariable();
        }


        printf("Fuzzing done==Tracker pwned!\n\n");
        return 0;
}

void printRequest(){
        //Skriver ut datan vi skickat.
        //Tömmer ev. gammal data.
        //Vi gör detta för att rädda terminalen.
        memset(buffer,0x00,sizeof(buffer));
        memcpy(buffer,s_get_databuf(),s_get_size());
        printf("Request:\n %.1500s\n",buffer);
}
//Bygger upp protokollet
void testData(){
/**
* Kod borttagen
**
}

//Filtrear bort ointressanta svar.
int filterBlankResponses(){

        //Hack (vill alltid skriva ut första svaret!
        if(!first){
                return 0;
        }

        char* buf = strstr(responseBuffer,"Content-Length: ");
        if(!buf){
                return 1;
        }
        char* res = strstr(buf, ": ");
        if (!res){
                return 1;
        }

        //Filterar ut tomma svar
        current_resp_len = atoi(&res[2]);


        if(current_resp_len > 0){

        }else{
                return 1;
        }



        if(DROP_SAME){
        //Filtrerar ut samma medelande om och om igen
        if ( strcmp(responseBuffer,prev_responseBuffer) == 0){
                return 1;
        }else{
                //Sparar buffern
                strncpy(prev_responseBuffer,responseBuffer,150000);
                }
```

```
        }


        //Filtrerar bort korrekta svar
        char *payload = strstr(responseBuffer,"\r\n\r\n");
        if( strcmp(correctResp,payload) == 0){
                return 1;
        }


        return 0;
}

/**
* Kod borttagen
**/
```

www.kth.se