# Fuzzing

En analys av fuzzingverktyget JBroFuzz

OSKAR LINDSTRÖM
och ALEXANDER MALMSTEDT


KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2010

# Fuzzing

En analys av fuzzingverktyget JBroFuzz

O S K A R  L I N D S T R Ö M
o c h  A L E X A N D E R  M A L M S T E D T

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

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/
lindstrom_oskar_OCH_malmstedt_alexander_K10001.pdf

Kungliga tekniska högskolan
*Skolan för datavetenskap och kommunikation*

**KTH** CSC
100 44 Stockholm

URL: www.kth.se/csc

**Abstract**

The purpose of this essay is to analyze a security tool called JBroFuzz, which will be used to examine a method of security testing called fuzzing. JBroFuzz will be used to test a range of fuzzing attacks against a web server. In this essay we will try to answer questions such as: What are the trademarks of a good fuzzer, how can JBroFuzz be used against a web server, is it possible to crash one of the world's most popular web servers using fuzzing and what should one have in mind when conducting tests in the future. Among our results, we have found that a good fuzzer should be able to support a variety of different fuzzing attacks, but also be able to carry out the attacks in parallel manner in order to test what happens when a web server has to handle a lot of requests at the same time. We have also found that JBroFuzz is not able to send parallel attacks, and therefore isn't a very great fuzzer. In future attempts, one should probably use a different fuzzer in order to harness the full power of fuzzing.

**Sammanfattning**

Syftet med den är uppsatsen är att analysera ett säkerhetsverktyg vid namn JBroFuzz, som kommer att användas för att utvärdera en metod inom säkerhetstestning som kallas fuzzing. JBroFuzz kommer att användas för att testa en rad olika fuzzingmetoder mot en webbserver. I den här uppsatsen kommer vi försöka svara på frågor som: Vilka egenskaper bör en bra fuzzer ha, hur kan JBroFuzz användas mot en webbserver, är det möjligt att krascha en av världens mest populära webbservrar med fuzzing samt vad ska man ha i åtanke för vidare experiment i framtiden. Under arbetet har vi kommit fram till att en bra fuzzer bör ha stöd för många olika sorters fuzzingmetoder, samt kunna utföra sina attacker parallellt för att undersöka vad som händer när en webbserver måste hantera många HTTP-förfrågningar samtidigt. Vi har också kommit fram till att JBroFuzz inte kan utföra sina attacker parallellt, och är därmed inte en särskilt bra fuzzer. I framtiden borde man troligtvis använda sig av en annan fuzzer för att kunna utnyttja den fulla potentialen av fuzzing.

# Innehåll

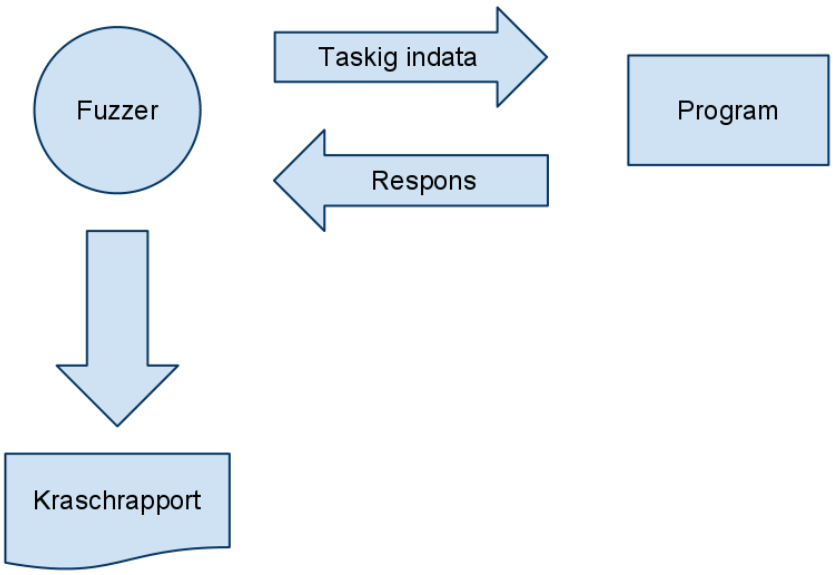
# Ordlista

- **Apache** - Ett företag som utvecklar Apache HTTP Server, en webbserver med fokus på robusthet och funktionalitet. [Foua]
- **ArchLinux** - En utgåva av operativsystemet Linux, community-drivet. [Arc]
- **Fuzzing** - En teknik inom mjukvarutestning som går ut på att skicka konstig indata till program. [Wika]
- **HTTP-förfrågan** - En förfrågan om information eller data som skickas mellan klient och server. [Wikb]
- **JBroFuzz** - En fuzzer som används för webbapplikationer. [Pav]
- **MySQL-injection** - En teknik för att utnyttja säkerhetsrisker i MySQL-databaser. [Wikd]
- **OWASP** - Stiftelsen som utvecklar JBroFuzz. [OWA]
- **Peach** - En annan fuzzer som kan utföra bl.a. mutationsbaserad fuzzing. [Edd]
- **Slowloris** - Ett program som kan skicka halvfärdiga HTTP-förfrågningar till webbservrar. [RSn]

# Kapitel 1

# Inledning

## 1.1 Bakgrund om fuzzing




Fuzzing är en teknik som går ut på att skicka konstiga eller rent av taskiga data till ett program för att sedan se vad det svarar. Det man är ute efter är att försöka krascha programmet, vilket betyder att man har hittat en bugg som måste åtgärdas.

Idén bakom fuzzing uppkom då professor Barton P. Miller satt på en uppringningsanslutning under en natt då det var åskväder och det dök upp konstiga tecken på hans skärm. Detta gav uppslag till en uppgift att skriva en fuzzer som genererade

konstiga tecken och skickade dessa till så många UNIX-program som möjligt (t.ex. grep, sed och uniq) för att se vad de skulle svara. Det visade sig att 25-33% av programmen kraschade. [Mil90]

Varför har inte dessa buggar hittas tidigare? Antagligen för att program snabbt växer i storlek och komplexitet, vilket gör dem svåra att överskåda. Detta är den kommersiella motivationen att använda fuzzing som ett testhjälpmedel under utvecklingen. Att tidigt hitta och rätta till buggar är till stor fördel då företaget kan undvika stora patchar och eventuella programkrascher.

## 1.2 JBroFuzz - programvaran vi testat

JBroFuzz är en fuzzer mot webbservrar och de välkända HTTP- och HTTPS-protokollet. Målet är att tillhandahålla ett enda portabelt program för fuzzing mot webbprotokollet.

JBroFuzz genererar en HTTP-förfrågan och skickar iväg den till den givna adressen. Sedan tar den emot svaret och sparar det. Den försöker inte på något sätt upptäcka om en sida är sårbar eller ej, detta är upp till det mänskliga ögat. [Pav]

## 1.3 Bakgrund om internetprotokoll

### 1.3.1 TCP-protokollet

TCP (Transmission Control Protocol) är ett protokoll som är optimerat för exakta dataöverföringar. Det garanterar en pålitlig leverans av data i rätt ordning från ett program på en dator till ett annat program på en annan dator.

Data skickas över internet i små paket, adresserade enligt IP-protokollet. Större datamängder delas upp hos avsändaren och sätts ihop hos mottagaren. Denna verksamhet följer TCP-protokollet, varav beteckningen TCP/IP för nättrafik i allmänhet.

Hur snabb överföringen kan bli med TCP/IP bestäms för varje förbindelse med en algoritm kallad *slow-start*. Den börjar med att skicka paket långsamt för att sedan öka exponentiellt tills paket börjar gå förlorade och då börjar den med diverse finjusteringar för att verkligen hitta den optimala överföringshastigheten. [Wikc]

### 1.3.2 HTTP-protokollet

TCP-protokollet används av både HTTP/1.0 och HTTP/1.1. [Nie97]

#### HTTP/1.0

För varje objekt, t.ex. textdokument eller bilder, som ska överföras öppnas en ny TCP-anslutning som sedan stängs då överföringen är klar. Att öppna en ny anslutning är långsamt i flera aspekter. För det första används slow-start som algoritm

för överföringshastigheten. Vidare kommer det skickas flera TCP-kontrollpaket innan objektet kan överföras. [Nie97] Eftersom internet mestadels består av många små objekt leder detta till att det blir många förfrågningar på små objekt. Vilket i sin tur leder till att många TCP-anslutningar måste öppnas och stängas. Detta är långsamt och nätverket kommer till stor del att bestå av TCP-protokollets kontrollpaket. Eftersom objekten är små hinner inte heller TCP-protokollets algoritm slutföras för att hitta den optimala överföringshastigheten. Detta betyder att större delen av objekten kommer överföras i en långsammare hastighet än vad som är möjligt och med mer nätverkstrafik på grund av de många förfrågningarna. [Nie97]

#### HTTP/1.1

Bakåtkompatibelt med HTTP/1.0 men med den stora skillnaden att den håller en TCP-anslutning öppen mellan efterföljande begäran (persistent connections). Fördelen med det är att man inte behöver öppna och stänga anslutningar samt att påverkan från slow-start inte blir lika stor, detta gör det mer effektivt. [Nie97]

Det finns även något som kallas ”pipelining” i HTTP. Det är en teknik som möjliggör att man kan skicka flera förfrågningar innan man har fått svar på dem och även få flera svar tillbaka samtidigt. Tack vare detta kan man baka in flera förfrågningar i samma TCP-segment med hjälp av att data buffras innan det sänds iväg. Detta drar i allmänhet ner antal paket och ökar prestandan ytterligare. [Nie97]

HTTP/1.1 stödjer också komprimering av överföringsdata, intervallbegäran och Cache-validering. Cache-validering tillsammans med intervallbegäran tros bli ett idiom för HTTP/1.1. [Nie97]

## 1.4 Apache webbserver - vårt angreppsmål

Världens vanligaste webbserver heter Apache. Det är en robust och funktionsrik HTTP-server med öppen källkod. [Foua]

Apache-projektets första implementation av HTTP-servern kom ut i april 1995. Den var byggd av buggfixar ovanpå NCSA HTTPd 1.3. Apache blev en stor succé och det tog mindre än ett år för den att bli världens mest använda HTTP-server. [Foua]

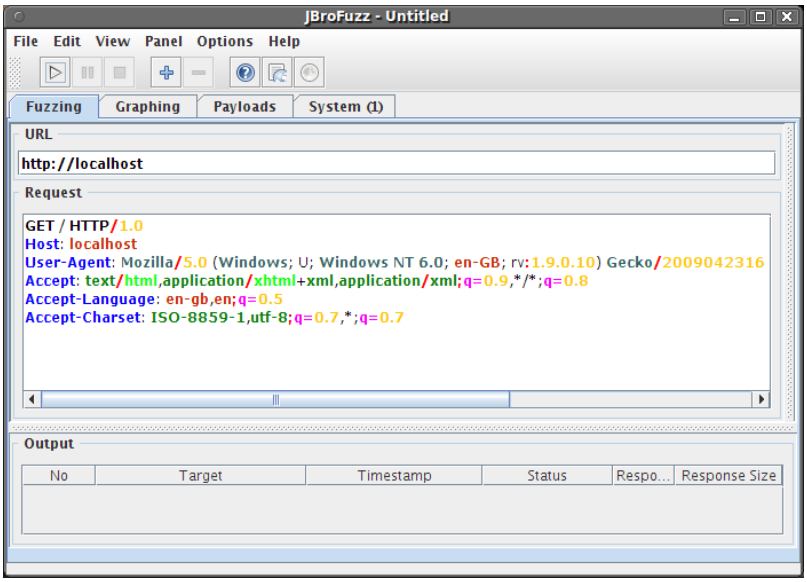
Namnet är valt för att hedra indianernas stam Apache, de ansågs vara överlägsna i krigslist och hade stor uthållighet. En annan mer populär historia är att namnet kommer från ”a patchy server”, eftersom den uppkom ur en patchad NCSA HTTPd-server. Dock är den sistnämnda officiellt fel. [Foub]

Apache Software Foundation försöker eliminera säkerhetsrisker och DOS-attacker mot Apache. Men det går inte att förhindra ett högt dataflöde till servern eller upprepade förfrågningar för samma sida. Istället har de en generell filosofi att undvika attacker som kan få servern att använda mer resurser än vad som behövs för att bearbeta data som kommer in. [Fouc]

# Kapitel 2

# Experiment

I vår undersökning kommer vi att utsätta en webbserver för fuzzing. Målet är att på några tänkbara sätt attackera webbservern med en kaskad av olika fuzzingmetoder, tills dess att kapitulationen är oundviklig. Angrepp kommer att ske på vissa fronter med förhoppningar om att stöta på serverkraschar, buggar, och konstiga felmeddelanden. Då vi inte med gott samvete kan ge oss på en godtycklig webbserver i denna kamp för bättre förståelse väljer vi att sätta upp vår egen webbserver att arbeta mot. Experimentet förbereddes genom installation av den nödvändiga mjukvaran. På klientdatorerna installerades JBroFuzz version 1.9 under operativsystemet ArchLinux, kernelversion 2.6.30. På serversidan installerades Apache-version 2.2.14 (Unix) under operativsystemet ArchLinux, kernelversion 2.6.32.




**Figur 2.1.** Huvudfönster för JBroFuzz.

## 2.1 Fuzzing av HTTP-protokollsversion

### Utförande

Den första metoden går ut på att modifiera HTTP-förfrågan som skickas till webbservern på ett sådant sätt att versionsnumret ändras på olika sätt. Tanken bakom den här tekniken är att undersöka hur webbservern svarar på ovanliga och konstigt formade förfrågningar. Att modifiera HTTP-protokollversionen kontrollerar hur webbservern väljer att behandla gamla protokoll, samt vad som händer om webbservern anropas med en protokollversion som inte existerar. En typisk HTTP-förfrågan med protokollversion 1.0 till en webbserver kan se ut såhär:

```
GET / HTTP/1.0
Host: localhost
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-GB; rv:1.9.0.10)
                    Gecko/2009042316 Firefox/3.0.10 (.NET CLR 3.5.30729)
                    JBroFuzz/1.9
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-gb,en;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

Med hjälp av JBroFuzz kan vi generera förfrågningar för protokollversioner mellan 0.0 och 1.9. I JBroFuzz's huvudfönster (Figur 2.1) kan vi modifiera förfrågningarna och lägga till olika fuzzers som genererar siffror. Först ändrar vi raden:

```
GET / HTTP/1.0
```

till:

```
GET / HTTP/0.0
```

Sedan markerar vi den sista nollan på raden och väljer att lägga till en fuzzer. En fuzzer är en slags modul i JBroFuzz som kan användas för att generera olika former av indata, som till exempel sifferserier, långa strängar, buffer overflows och MySQL-injections. Vi väljer att lägga till en "Base 10 (Decimal) Alphabet"-fuzzer som kommer att iterera den markerade siffran mellan 0 och 9. När vi sedan klickar på start skickar JBroFuzz ut 10 HTTP-förfrågningar till webbservern, där protokollversionen går från 0.0 upp till 0.9 och returnerar resultat för varje förfrågan. Sedan markerar vi den vänstra siffran och ändrar den från en nolla till en etta. När vi återigen klickar på start kommer nu JBroFuzz att skicka 10 nya förfrågningar med versionsnummer 1.0 t.o.m 1.9.

### Resultat

Svaren från varje enskild förfrågan undersöks i JBroFuzz's huvudfönster, där man bland annat kan få information om responstiden för varje förfrågan. Under testet märkte vi en väldigt stor skillnad i responstiden för de olika protokollversionerna.

För protokollversion 0.0 upp till 1.1 låg genomsnittstiden på cirka 21 millisekunder. För de efterföljande protokollversionerna 1.2 upp till 1.9 ökade responstiden för servern till cirka 5000 millisekunder. Ingen av anropen skapade någon form av krasch eller fel hos servern, och varje förfrågan fick en korrekt respons av servern.

### Analys

En varierande responstid kan ha många olika förklaringar, dels kan det bara vara enstaka störningar i överföringen, dels kan det vara så att förfrågningarna faktiskt behandlas på olika sätt. I detta fall tror vi mer på det senare, då förändringen i responstid var både stor och konsekvent för upprepade försök. Troligtvis behandlas förfrågningar med anrop till framtida HTTP-versioner på ett helt annat sätt än en vanlig förfrågan. Vilken konsekvens det får i verkligheten är svårt att säga, då man inte kan förvänta sig att webbservrar ska kunna stödja den versionen ännu, men är absolut något som bör undersökas närmare då det kan få konsekvenser när det väl är dags att byta protokollversion. Att förfrågningar för oimplementerade protokoll tar längre tid än vanliga förfrågningar är något förbryllande, men inget som vi kan förklara utan en noggrann analys av Apaches källkod.

## 2.2 Fuzzing av förfrågningsmetoder

### Utförande

Den andra metoden liknar den första metoden på många vis. Istället för protokollversionen kommer nu HTTP-förfrågningsmetoden att genereras på olika sätt. De vanligaste förfrågningsmetoderna i modern tid är POST samt GET. Av bakåtkompabilitetsskäl finns det fortfarande många äldre förfrågningsmetoder kvar i HTTP-protokollet som till exempel PUT, DELETE och MOVE. Vi modifierar den ursprungliga HTTP-förfrågan genom att lägga till en ”HTTP methods”-fuzzer. Denna fuzzer genererar samtliga förfrågningsmetoder som är tillgängliga i JBroFuzz. För just denna fuzzer måste vi även välja encoding till ”Uppercase” för att generera metoderna på rätt sätt. När vi klickar på start skickar JBroFuzz ut 15 stycken förfrågningar med de utvalda förfrågningsmetoderna.

### Resultat

Som förväntat rasslar den vanliga GET-metoden igenom utan problem. Likaså fungerar HEAD-metoden, som ju enbart returnerar headers, men därefter uppstår komplikationer. För samtliga övriga metoder, förutom metoden SEARCH som inte fick något svar på grund av att det trådlösa nätverket gick ned, får vi felmeddelandet ”404 - bad request” efter en responstid på 300000 millisekunder, d.v.s 5 minuter! Efter en tids konfundering insåg vi att detta är standardinställningen för timeouttiden på en socket för en Apache webbserver. Med andra ord ligger förfrågan kvar i 5 minuter och upptar resurser på serversidan, innan webbservern väljer att terminera

den. Inte heller i detta experiment orsakade något av anropen någon form av krasch hos servern.

### Analys

Att de flesta av förfrågningarna får en timeout är inte särskilt oroväckande, då vi faktiskt gör felaktiga anrop till en webbserver. Webbservern har inget val förutom att avbryta anslutningen efter ett tag, då vi inte skickar den data som servern behöver. Något som är något mer oroväckande är dock den otroligt långa responstiden för dessa förfrågningar. En timeout på 300 sekunder som standard verkar något stort, men är faktiskt något som behövs för t.ex. passiva filuppladdningar, som tar lång tid. Att skapa många sådana här förfrågningar parallellt hade antagligen orsakat en del problem hos webbservern, då detta upptar resurser. Dessvärre kan vi inte testa det i detta experiment, då detta ej stöds av JBroFuzz.

## 2.3 Fuzzing med anrop till ogiltiga sidor

### Utförande

I detta försök kommer vi att undersöka hur webbservern reagerar på anrop till sidor som inte existerar. Förutom att se reaktionen från webbservern är det även möjligt att hitta sidor som användaren egentligen inte är menad att kika på, en så kallad ”bruteforce”-attack. Slutligen är detta även en metod för att försöka skapa ”buffer overflows” hos servern genom att försöka komma åt sidor med väldigt långa namn. Vi utför detta experiment genom att markera positionen precis bakom ”/”-et i förfrågan:

```
GET / HTTP/1.0
```

för att sedan lägga till fuzzern ”Long string of aaa’s”.

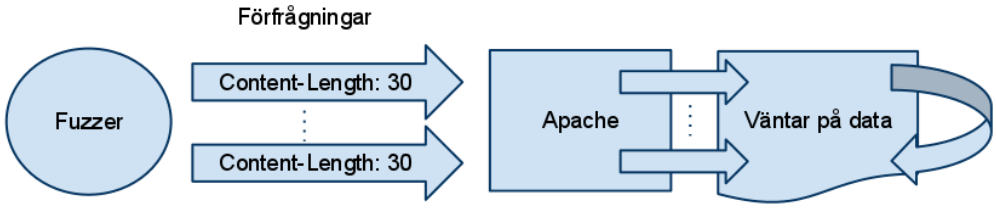
### Resultat

När vi nu startar angreppet kommer JBroFuzz att generera filnamn av olika storlekar och sedan göra HTTP-förfrågningar på dessa filer till webbservern. Resultatet blir något märkligt. För filnamn på en längd av ett a upp till 129 a:n svarar servern med ”404 - Page not found”, ett förväntat svar då dessa sidor inte existerar på webbservern. Vid nästa försök, med 257 a:n, svarar dock servern ”403 - Forbidden”. Detta sker även för filnamn på upp till 4197 a:n. Vid ännu större filnamn ger servern återigen ett nytt svar, nämligen ”414 - Request-URI too large”. Detta meddelande visades för alla resterande testade filnamnslängder, där den största storleken vi testade var 65536 a:n.

### Analys

Svaren som webbservern ger oss leder till förvirring. När man försöker hämta hem sidor som inte finns känns det naturligt att få svaret att sidan inte finns, men däremot att få svaret ”forbidden” för tillräckligt långa namn verkar något märkligt. Apache har av någon anledning valt att hantera olika långa namn på olika sätt. Detta kan med viss sannolikhet vara en bugg hos Apache, då vi inte kan finna någon speciell anledning till att behandla förfrågningar på detta sätt. Att vi till slut får svaret ”Request-URI too large” för extremt stora filnamn känns också rätt, då man annars skulle kunna orsaka ”buffer overflows” med sådana namn. Att intervallet däremellan specialbehandlas skulle mycket väl kunna utnyttjas till att bilda någon form av säkerhetsrisk hos webbservern, om vi hade haft mer tid till att fuzza servern.

## 2.4 Fuzzing med ofärdiga förfrågningar




**Figur 2.2.** *Tanken bakom ofärdiga förfrågningar är att de ska ligga och ta upp resurser på serversidan.*

### Utförande

I den fjärde metoden kommer vi att undersöka vad som händer om man skickar ofärdiga HTTP-förfrågningar till en webbserver. I just detta försök säger vi åt webbservern att den kan förvänta sig data av en viss längd, för att sedan strunta i att skicka själva datat. För att utföra detta lägger vi till raden ”Content-Length: 30” på sista raden i vår HTTP-förfrågan. Detta innebär att webbservern förväntar sig att 30 byte data ska skickas med i förfrågan. Sedan lägger vi till fuzzern ”100 plain requests” som ser till att vi skickar vår förfrågan 100 gånger.

### Resultat

I likhet med experimentet med förfrågningsmetoder fick vi även här en väldigt lång responstid på förfrågningarna, innan servern till slut svarade med ”400 - bad request” efter cirka 5 minuter. Med andra ord låg återigen förfrågningarna kvar och upptog resurser på serversidan, tills dess att serversidan terminerade anslutningen.

### Analys

Då vi fortfarande ej har möjlighet att skicka parallella förfrågninar till webbservern, så kan vi inte undersöka hur pass mycket resurser dessa förfrågningar upptar. Att skicka dessa typer av förfrågningar kan dock ha stor påverkan på webbservern, då vi faktiskt lovar att vi kommer att skicka data, vilket antagligen kommer få webbservern att allokera plats för den data som snart komma skall.

## 2.5 Fuzzing med upprepade förfrågningar

### Utförande

Ett ytterligare sätt att attackera en webbserver på är att skicka upprepade förfrågningar i en snabb följd för att se hur webbservern klarar av det. Med hjälp av detta kan man undersöka saker som hur många anslutningar servern klarar av samtidigt, hur mycket data den kan hålla reda på samtidigt, eller hur snabbt servern kan arbeta vid hög belastning. För att testa detta använder vi oss av vår ursprungliga förfrågan som förväntas få ett vanligt ”200 - OK” svar eftersom att förfrågan är giltig. Sedan lägger vi till fuzzern ”10000 plain requests” vilket kommer, som namnet antyder, att skicka förfrågan 10000 gånger.

### Resultat

Förfrågningarna flyger iväg i en rasande fart, men det tar ändå flera minuter innan vi får svar på alla. Samtliga kommer dock tillbaka med det förväntade svaret ”200 - OK”. Den genomsnittsliga responstiden för en förfrågan ligger ganska stabilt på cirka 20 millisekunder, med vissa avstickare på 3000 millisekunder, och även en riktig slö förfrågan som tog 21000 millisekunder. Servern verkade för övrigt inte särskilt påverkad av fuzzern, då det var möjligt att komma åt den som vanligt via en webbläsare under tiden som förfrågningarna skickades.

### Analys

Att webbservern inte påverkas särskilt i detta fallet beror troligtvis på att JBroFuzz inte kan skicka sina förfrågningar parallellt. De enstaka (färre än 10) förfrågningar som hade mycket längre responstid än de andra beror med stor sannolikhet på den halvstabila trådlösa anslutningen som vi använder oss av. Med andra ord hade detta experiment givet mer intressanta resultat om vi hade kunnat använda oss av ett verktyg med stöd för parallella förfrågningar.

# Kapitel 3

# Diskussion

## 3.1 Förhållanden för experimenten

När vi utförde experimenten märkte vi snart en begränsning hos JBroFuzz, nämligen att programmet skickade sina HTTP-förfrågningar seriellt till webbservern. Detta ledde dels till att vi var tvungna att vänta väldigt länge på de förfrågningar som tog lång tid på sig, dels till att vi antagligen inte påverkade prestandan hos webbservern tillräckligt mycket för att se några uppenbara resultat.

## 3.2 Våra felkällor

### Belastning på servern

Servern vi har utfört alla experiment emot är inte enbart en Apacheserver, utan har även en ssh daemon som enligt loggfilen (/var/log/auth.log) får ca 30 000 anslutningar varje dag. Detta innebär att linan delas med en drös av folk som genom brute force försöker få kontroll över datorn. Detta stjäl både processorcykler och bandbredd, dock inte i en utsträckning att det märkbart påverkar resultaten. Vi har inte heller för avsikt att undersöka Apaches robusthet, utan snarare att analysera JBroFuzz.

### Trådlösa nätverk

Ytterligare felkällor är KTHs trådlösa nätverk som vi inte har någon kontroll över. Där kan man plötsligt bli frånkopplad, för att två sekunder senare vara ansluten igen.

Det trådlösa nätverk på den dator som vi har testat från är något instabilt och det händer ibland att det plötsligt dör. För att sedan starta det igen måste nätverkskortet tas ner och sen upp, ibland till och med starta om hela datorn. Den gång det har inträffat har det redovisats i resultatet.

# Kapitel 4

# Slutsatser

## 4.1 Den ultimata Fuzzern

Vid utförandet av dessa experiment har vi hittat egenskaper vi både gillat och saknat hos JBroFuzz. Nedan följer hur vi vill ha vår ultimata fuzzer.

### Parallella förfrågningar

Vi anser att en fuzzer bör ha ett val om man vill ha förfrågningar skickade parallellt eller seriellt, då vi gärna hade testat hur Apache hanterar flera förfrågningar samtidigt. Kanske händer det något internt som skulle kunna ge intressanta svar, som hur hanterar Apache inkommande anslutningar när trådpoolen för dem är full t.ex.

Det vore också bra om man kunde välja att ta bort de parallella förfrågningarna om man exempelvis har lyckats krascha ett program och skall ta reda på exakt varför det kraschade. Då är det bra att skicka data seriellt, ty då vet man i vilken ordning data kommer fram, något som blir väldigt mycket svårare om man kör parallellt.

### Användargränssnitt

En viktig del när det gäller datorprogram i allmänhet är användargränssnittet. Ett användargränssnitt bör vara intuitivt och lätt att förstå. Det bör ge en bra överblick, men samtidigt ge tillräckligt detaljerad information när så efterlängtas. Dessa egenskaper är även eftertraktade i en bra fuzzer, då man gärna vill kunna se detaljerad information om varje förfrågan på ett smidigt sätt. Dessvärre fallerar JBroFuzz en del på den punkten, då man måste högerklicka på varje enskilt svar för att kunna få ut någon relevant information från dem. En enkel ”expand all”-knapp som visar resultaten från alla svar samtidigt hade ökat användarvänligheten avsevärt.

En annan del av användargränssnittet som JBroFuzz bör bättra på är markering av var man har lagt till en fuzzingmodul i sin förfrågan. Just nu markerar man enbart

en position i förfrågan och lägger till en modul, men därefter kan vi inte se var vi har lagt ut modulerna. Det enda sättet att se om en förfrågan kommer att se ut som man föreställt sig är att skicka iväg förfrågan, vänta på svaret, högerklicka svaret och sedan kolla på den skickade informationen.

Ytterligare en viktig del när det gäller gränssnittet är att presentera resultatet på ett användbart sätt. Förvisso kan man ta reda på det mesta enbart genom att undersöka varje enskilt resultat, men en bra fuzzer bör kunna generera grafer och dylikt som presenterar resultat på ett trevligt sätt. Lyckligtvis är JBroFuzz väldigt bra på denna punkt, då det är möjligt att markera resultat och generera grafer utifrån dem, som kan visa fördelningen av responstider, svarstyper etc. Det är även möjligt att högerklicka på graferna och spara dem som bilder på en gång, vilket är väldigt smidigt.

### Moduler

En av de absolut viktigaste delarna hos en fuzzer är ju så klart själva fuzzingen. En omfattande mängd moduler, som kan fuzza på en stor variation av olika sätt är därför centralt hos en fuzzer. Styrkan i en fuzzer ligger i att ha så många förgjorda moduler som möjligt, för att avlasta tankearbetet för användaren som annars själv måste komma på konstiga indata som skulle kunna skickas. Å andra sidan är det minst lika viktigt att användaren själv kan lägga till sina egna moduler, då man vill fuzza på ett specifikt sätt. Detta har JBroFuzz lyckats ganska bra med, då det finns stöd för att lägga till egna moduler. Hur pass svårt detta är kan vi dock inte svara på, då vi ej har testat det. En annan fuzzer vid namn Peach har implementerat skapande av moduler med XML-filer, något som till första anblick verkar vara lättare att förstå än det format JBroFuzz använder sig av. Peach verkar dock köra på mottot ”Går det inte att lösa med XML så går det att lösa med mer XML” då varje modulfil verkar vara väldigt stor och komplicerad.

JBroFuzz har även en relativt stor mängd färdiga moduler att använda sig av, med allt från vanliga sifferserier till databasanrop. En bättre fuzzer hade dock haft lite fler inställningar för varje modul. I JBroFuzz är det i princip enbart möjligt att välja om modulen ska använda små eller stora bokstäver och liknande, det hade vart mycket bättre om man till exempel kunde skicka in bara udda nummer, bara primtal och liknande saker. Varje modul behöver även ha en förklaring vad den gör och vad den kan användas till, något som saknas i JBroFuzz. Här får man helt enkelt chansa lite.

En annan viktig egenskap hos en fuzzer är något som kallas för mutationsbaserad fuzzing, som innebär i korta drag att fuzzern kan utgå från en ”mall”, och baserat på svaren den får kan ändra mallen och ”mutera” den. Det är ett värdefullt hjälpmedel då det överlämnar mer av tänkandet åt själva fuzzern. Detta är något som finns implementerat i Peach, men som tyvärr inte verkar stödjas av JBroFuzz.

### Övriga egenskaper

En fuzzer ska även ha många val och inställningar så att man kan skicka data på precis det sättet man själv vill. Möjlighet till fininställningar bör också finnas i den perfekta fuzzern. Utöver detta tycker vi även att en fuzzer ska ha väl kännedom om det protokoll som den är avsedd att fuzza. Detta är en nödvändighet då man använder sig av mutationsbaserad fuzzing. Hur ska den annars veta hur den ska mutera sina förfrågan? Det skulle då vara möjligt att ge förslag på vilka fuzzingmetoder som är lämpliga för det man avser att fuzza.

En något primitiv egenskap är även att skicka slumpmässig data. Detta kan ha sina fördelar då den kan generera väldigt konstiga förfrågningar som man själv inte har tänkt på. Dock kan det ta väldigt lång tid innan just en sådan förfrågan generaras vilket är nackdelen med denna taktik.

Programmet borde även vara licensierat under någon öppen källkod. Då finns möjligheten att titta in i koden och se exakt hur den fungerar och man kan även själv ändra i koden för att lägga till önskad funktionalitet.

## 4.2 Idéer för framtida försök

Om man hade all tid i världen till fuzzing hade det varit intressant att jämföra resultatet från olika fuzzers mot samma angreppsmål. Från skillnaderna kan man förhoppningsvis utläsa intressanta mönster som leder en vidare.

Man bör även sätta sig in i källkoden för det man angriper, förutsatt att den finns tillgänglig. Med god insikt i hur den interna strukturen ser ut är det lättare att upptäcka nya angreppsmål och komma på nya sätt att applicera fuzzingmodulerna. När man hittat något man vill testa skriver man en egen modul för att göra det och på det sättet kan man bestämma exakt hur saker och ting ska angripas.

Man borde även testat flera webbservrar än bara Apache som vi gjorde. Apache har ju som sagt inriktat sig på just robusthet, så att testa en mindre populära webbserver hade kanske gett mer givande resultat. Vidare borde man undersöka på vilka sätt man kan skydda sig mot angrepp, genom inställningar i Apache t.ex. Hela poängen med fuzzing är ju att hitta buggar, så att experimentera med saker som timeout-tider och andra säkerhetsinställnigar för att förhindra dessa buggar är helt klart intressant.

Något vi inte heller testade var exakt vilken längd på filnamn som gav ”403 - Forbidden” (i 2.3 Fuzzing med anrop till ogiltiga sidor). Det var ett intressant svar och det borde tas reda på varför det blir på det sättet.

### Slowloris

Slowloris är ett perlscript skrivet av RSnake med syfte att skapa en denial of service attack med låg bandbredd. [RSn] Scriptet fungerar som så att den skickar halvfärdiga förfrågningar till HTTP-servern genom att sätta headern ”Content-Length” till 42, för att sedan aldrig skicka några data, precis som vi gjorde i 2.4. Detta gör att

servern ligger och väntar på data som den aldrig kommer att få. Inte nog med det skickas många ofärdiga förfrågningar samtidigt. Med hjälp av detta verktyg lyckades vi få trådpoolen för inkommande anslutningar att ta slut, vilket ledde till att det inte längre gick att komma åt webbservern genom en webbläsare. Detta fungerar eftersom Slowloris kan skicka sina förfrågningar parallellt.

**Avslutningsvis** skulle man kunna säga att många av våra resultat antagligen hade sett helt annorlunda ut om vi hade kunnat skicka våra förfrågningar parallellt. Dessvärre lyckades vi ju inte krascha webbservern, men har däremot fått en hel del nya tankar och idéer om hur man skulle kunna lyckas. Att det är möjligt att få servern att sluta svara med hjälp av Slowloris är ju faktiskt en rätt bra ledtråd till att alla säkerhetshål inte är helt tilltäppta i Apache än. Den långa responstiden som uppstår vid anrop med ogiltiga HTTP-versioner är något som absolut vore intressant att undersöka närmare i framtiden. Efter våra experiment känner vi att fuzzing är en bra hjälp när det gäller att hitta buggar, och är ett ämnesområde som alla säkerhetstestare borde ha kännedom om, och som troligtvis kommer växa stort med tiden.

# Litteraturförteckning


[Arc] ArchLinux. About the Archlinux. http://wiki.archlinux.org/index.php/Arch_Linux (2010-04-19).

[Edd] Michael Eddington. What is Peach? http://peachfuzzer.com/WhatIsPeach (2010-05-03).

[Foua] The Apache Software Foundation. About the Apache HTTP Server Project. http://httpd.apache.org/ABOUT_APACHE.html (2010-02-28).

[Foub] The Apache Software Foundation. Apache Server Frequently Asked Questions. http://httpd.apache.org/docs/1.3/misc/FAQ.html (2010-03-01).

[Fouc] The Apache Software Foundation. Reporting Security Problems with Apache. http://httpd.apache.org/security_report.html (2010-03-01).

[Mil90] Barton P. Miller. An Empirical Study of the Reliability of UNIX Utilities. ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf (2010-02-12), 1990.

[Nie97] Henrik Frystyk Nielsen. Network Performance Effects of HTTP/1.1, CSS1, and PNG. http://www.w3.org/Protocols/HTTP/Performance/Pipeline.html (2010-02-25), 1997.

[OWA] OWASP. About The Open Web Application Security Project. http://www.owasp.org/index.php/About_OWASP (2010-03-01).

[Pav] Yiannis Pavlosoglou. OWASP JBroFuzz. http://www.owasp.org/index.php/Category:OWASP_JBroFuzz (2010-03-01).

[RSn] RSnake. Slowloris HTTP DoS. http://ha.ckers.org/slowloris/ (2010-02-19).

[Wika] Wikipedia. Fuzz testing. http://en.wikipedia.org/wiki/Fuzz_testing (2010-02-11).

[Wikb] Wikipedia. Hypertext Transfer Protocol. http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol (2010-02-17).

[Wikc] Wikipedia. Slow-start. http://en.wikipedia.org/wiki/Slow-start (2010-02-25).

[Wikd] Wikipedia. SQL injection. http://en.wikipedia.org/wiki/SQL_injection (2010-04-19).