
KTH Computer Science
and Communication

# Domän-Webb-Applikations-Fuzzer (DWAF)

Introduktion och implementation

HANIF FARAHMAND MOKARREMI,
ASHKAN JAHANBAKHSH

Examensrapport vid NADA, DD143X, KTH
Handledare och examinator: Mads Dam

Datum 2011-06-14

# Referat

Fuzzing eller fuzz-testning är en automatiserad testningsmetod för datorprogram. Tekniken har av olika anledningar blivit allt vanligare som testningsmetod. Den här rapporten kommer först att beskriva varför det finns ett behov av en fuzzer som testar/kan testa flera applikationer i en domän. Den förklarar sedan de viktiga delarna i en fuzzer och hur man implementerar en sådan. Avslutningsvis testas den implementerade fuzzern DWAF mot ett antal webbapplikationer som tydlig visar att den har förmågan att hitta många SQL-injection och XSS-buggar.

# Abstract

## Domain-Webb-Applications-Fuzzer (DWAF)

Fuzzing, or fuzz testing is an automated testing technique for computer programs. For various reasons it has become increasingly common to use this technology. This report will first describe why there is a need for a fuzzer that can test several applications in a domain. Then, it explains the important elements of a fuzzer and how to implement them. Finally we will test our own implementation of a fuzzer, DWAF, on a number of web applications, which clearly shows that it is able to find many SQL-injections and XSS bugs.

# Statement of collaboration

Implementeringen av DWAF delades upp i två delar. En crawler-del och en fuzzer-del. Hanif kodade crawler-delen och Ashkan fuzzer-delen. Ashkan har också skrivit fuzzer-delen, sårbarhetstyperna, hanterat testerna och tagit bilder, medan Hanif har skrivit resterande och haft ansvar för LaTeX-formatering och rättstavningen.

# Innehåll

# Kapitel 1

# Introduktion

Fuzzing eller fuzz-testning är en automatiserad testningsmetod för datorprogram. Tekniken har av olika anledningar blivit allt vanligare som testningsmetod. Användning av fuzzing sträcker sig från filtyper och lokala applikationer till webbsidor och webbapplikationer. Rapport beskriver inledningsvis varför det finns ett behov av en fuzzer som testar/kan testa flera applikationer i en domän. Den förklara sedan de viktiga delarna i en fuzzer och hur man implementerar en sådan. Avslutningsvis implementeras webbapplikations-fuzzern DWAF och testas mot ett antal sidor för att redogöra för dess styrka och svagheter.

I ett senare skede kommer vi även att testa vår fuzzer mot CSCs Rapp-system. Anledning till detta är dels att Rapp-systemet är ganska nytt och därför kan innehålla en del buggar, men också för att bidra med information till KTH/CSC.

Rapporten behandlar följande frågeställningar:

- Varför det är viktigt med webbapplikationsfuzzer?
- Redogörelse för de olika fuzzing-metoderna.
- Vilka är de viktiga delarna i en fungerande fuzzer?
- Hur implementerar man en webbapplikationsfuzzer?

# Kapitel 2

# Webbapplikation

Webbapplikation är en applikation som är tillgänglig över ett nätverk som Internet eller ett intranät. Webbapplikationer är skapade på ett sätt som gör att de kan brukas av en användare via en webbsida i en webbläsare. Exempel på webbaplikationer kan vara webb-mail, sociala nätverk, e-banker, bloggar, webbtidningar och många andra dynamiska sidor på nätet. På senare tid har förväntningarna på användarvänligheten och funktionaliteten på många webbapplikationer ökat markant. T.ex. finns det olika sätt att skicka indata till webbapplikationer genom att endast skicka med dem som parametrar i ett adressfält i en webbläsare. Dessa inmatningsmöjligheter förenklar bl.a. delning av länkar mellan olika personer. Men dessa länkar kan lätt manipuleras och om de inte hanteras korrekt kan det i vissa fall resultera i säkerhetshål som kan utnyttjas av hackare. Det är just dessa säkerhetshål som är av intresse för den här rapporten.

# Kapitel 3

# Mjukvarutestning

## 3.1 Testningsmetoder

Man brukar dela upp testnings- och felsökningsmetoder i tre olika kategorier. Benämningen på dessa är som mycket annat i datorvärlden på engelska, och de heter White-box, Black-box och Gray-box fuzzing. Fuzzing som testningsmetod faller inom ramen av s.k. black-box och gray-box testning. Dock kan det vara av intresse att känna till övriga möjliga tillvägagångssätt.

White-box-testning innebär att man har full insyn i hur programmet (eller det man vill testa) är uppbyggt och har även tillgång till källkoden.

Ett exempel på White Box testning:

```
1 #include <string.h>
2 int main(int argc, char **argv)
3 {
4   char buffer[10];
5   strcpy(buffer, "test");
6 }
```

Kodstycket ovan visar en enkel kopiering av en string till en char-array med bestämd storlek.

```
1 #include <string.h>
2 int main(int argc, char **argv)
3 {
4   char buffer[10];
5   strcpy(buffer, argv[1]);
6 }
```

I det här kodstycket byts strängen ut mot användarens egen inmatning. Här kan vi direkt se att om användaren matar in en sträng som har längden större än 10 så kraschar programmet.

En typ av White-box testning är “Enhetstestning” (fr. eng. Unit Testing) som är en metod som testar enskilda enheter av källkod för att avgöra om de är lämpliga för användning. En enhet är den minsta testbara del av ett program. I procedurell programmering kan en enhet vara en enskild funktion eller procedur. I objektorienterad programmering är en enhet vanligtvis en metod. Enhetstester skapas vanligtvis av programmerare eller ibland av White-box testare under utvecklingsprocessen [26].

Black-box testning innebär att man, i motsats till White-box testning, inte har tillgång till någon sorts information. Man vet inte hur programmet är uppbyggt och inte heller vad den kan ta som indata. Det enda man har kunskap om är det som man kan observera. Användaren kan själv bestämma vad den vill skicka som indata till programmet och man kan sedan observera utdatan. En typ av Black-box testning är “Systemtestning” (fr. eng. System testing) som är en metod som avser testa hela system eller färdiga produkter för att avgöra om de uppfyller de förbestämda specifikationskraven [25].

Gray-box är ett begrepp som varken är White- eller Black-box. Det brukar användas vid de situationer då man har tillgång till information om hur applikationen är uppbyggd och vilka indata den kan ta emot. Man har dock i allmänhet inte tillgång till källkoden.

Antagandet att White-box-testning skulle vara ett bättre eller effektivare tillvägagångssätt, för att man har tillgång till källkoden, är inte riktigt sant. Det är viktigt att poängtera att det som man ser i källkoden inte alltid är det som man får ut när man exekverar programmet. T.ex. kan kompileringsvektygen göra ganska drastiska ändringar i koden. De olika testningsvarianterna kan snarare ses som diverse testningsmetoder än att det ena skulle vara bättre än det andra [11, p. 9].

Tillgängliga metoder för White-box testning är att antingen gå igenom koden manuellt eller att låta en automat ta hand om det. Då en källkod kan innehålla massvis med rader kod, är det första alternativet nästan helt uteslutet för alla större program. För Black- och Gray-box kan man utveckla en fuzzer, vilket förklaras utförligt i nästa kapitel.

# Kapitel 4

# Fuzzer

Fuzzing är en testningsmetod som går ut på att hel- eller semiautomatiskt mata in ogiltig och/eller stora mängder indata till en applikation för att sedan analysera hur den beter sig för de olika inmatningarna. Syftet med fuzzing är att få en applikation att krascha eller bete sig på ett sätt som den inte är ämnat för. Begreppet fuzzing härstammar från en studie av professor Barton Miller där han år 1988 tillsammans med sina elever testade alla unix kommandon med slumpmässiga data och fick 25-33% av dem att krascha [22].

Fuzzer-programmen faller inom två kategorier. De mutation-baserade (fr. eng. mutation-based) som avser att ändra redan existerande indata för att skapa nya testscenarion, och de generation-baserade (fr. eng. generation-based) som avser att skapa testdata baserad på modeller och kunskap om indata.

Den enklaste formen av fuzzing teknik är att skicka en ström av slumpmässiga indata till en programvara, antingen som kommandoradsflaggor, slumpmässigt muterade protokollpaket eller som händelser. Denna tekniken är ett kraftfullt verktyg för att hitta fel i kommandoprompt-program, nätverksprotokoll, och GUI-baserade applikationer och tjänster.

En annan vanlig teknik som är enkel att genomföra är att mutera befintlig indata genom att ändra/flytta runt bitarna på måfå eller flytta runt delar av en fil. De mest framgångsrika fuzzer-applikationer bygger däremot på detaljerad förståelse av formatet eller protokollet som avses testas. Förståelsen kan vara baserad på formatets eller protokollets specifikation eller rent empiriskt information om tidigare funna buggar och svagheter i det specifika systemet.

## 4.1 Fuzzingtyper

Olika fuzzingtyper är baserade på målet det vill säga olika mål har olika egenskaper och varje typ lämpar sig för sin egen klass av fuzzer. Nedan förklaras olika fuzzingtyper och

deras användningsområden.

### 4.1.1 Lokala fuzzers

I UNIX-världen tillåter setuid applikationer vanliga användare att tillfälligt kunna höja sina rättighetsnivåer i operativ systemet. Just därför är setuid-applikationer ett självklart mål för fuzzning. Någon sorts av bugg kan leda till rotåtkomst för angriparen. Det finns två olika typer av lokala fuzzers:

- kommandotolk-baserad fuzzing, där testdata skickas till setuid-applikationen genom kommandotolken.
- miljövariabel fuzzing, där test-datan skickas genom UNIX-skal (fr. eng. shell) miljön.

### 4.1.2 Fjärr-fuzzers

Målen för fjärr-fuzzers är applikationer som lyssnar på en nätverksgränssnitt. Olika tjänster på Internet så som hemsidor, e-mail, olika nätverksprotokoller har en stor roll för användaren av Internet. En bugg i vilket som helst av dessa tjänster kan ge möjlighet till en angripare att stjäla värdefull information från de drabbade systemen.

## 4.2 Sårbarhetstyper

För att kunna framgångsrikt fuzza en webbapplikation är det viktigt att veta vilka olika sårbarhetstyper en webbapplikation kan ha. I de kommande styckena kommer det att redogöras för de mest förkommande sårbarhetstyperna.

### 4.2.1 Local File Inclusion (LFI)

Local File Inclusion eller LFI [4] är en metod för att utnyttja en webbapplikation till att läsa andra filer från servern genom t.ex. en webbläsare. Den här sårbarhetstypen kan utnyttjas när utvecklaren använder sig av GET requests för att medta (fr. eng. include) filer från servern.

Ett typiskt exempel på ett sårbart PHP-skript:

```
1  <?php
2      $file = $_GET['file'];
3      if(isset($file))
4      {
5          include("pages/$file");
6      }
7      else
8      {
9          include("index.php");
10     }
11 ?>
```

Ett giltig begäran till skriptet kan vara:

```
1 http://target.com/index.php?file=file.php
```

En angripare kan då ändra på detta och försöka se innehållet på andra filer i servern:

```
1 http://target.com/index.php?file=../../../../../etc/passwd%00
```

Vilket i det här fallet tillåter angriparen att se innehållet på filen /etc/passwd i en *NIX server.

### 4.2.2 Cross Site Scripting (XSS)

Cross Site Scripting eller XSS [21] handlar om att bädda in HTML- eller Javascriptkod i en hemsidas indata och på så sätt ändra sidans funktionaliteten efter angriparens behov. Ett enkelt exempel är en sökmotor. När sökresultaten visas, brukar även söktermen visas. ett typexempel kan vara så här i sidans källkod:

```
1 <html>
2 <body>
3 ...
4 <p>soktermen</p>
5 ...
6 </body>
7 </html>
```

En angripare kan då ersätta söktermen med valfri javascript. I detta exempel ersätter vi det med följande kod:

```
1 <script>alert('XSS');</script>
```

Vi får då följande kod i webbsidans källkod:

```
1 <html>
2 <body>
3 ...
4 <p><script>alert('XSS');</script></p>
5 ...
6 </body>
7 </html>
```

Efter exekveringen visas ett meddelande med “XSS” på skärmen. Angriparen kan utnyttja säkerhetshål som detta genom att skicka s.k. kakor (fr. eng. cookies) och vidare kunna använda dessa för att logga in eller stjäla data från den drabbade.

### 4.2.3 SQL-injection

SQL-injection [24] inträffar när en webbapplikation accepterar användardata som direkt placeras i en SQL-sats och inte korrekt filtrerar bort farliga/ogiltiga tecken. Detta kan ge

angripare åtkomst till serverns databas och tillåta denna att inte bara stjäla data från databasen, utan även ändra och ta bort data.

Nedanstående SQL-kommando är ett exempel på ett SQL-injection.

```
1 select * from users where username ='" & username & "' and password =' "
    & password & "'
```

Om man då tänker sig att en angripare vet att en användare har användarnamnet admin, men inte vet lösenordet kan då angriparen mata in följande indatan:

```
1 username = "admin"
2 password = "' or '1'='1"
```

Detta kommer innebära att angriparen blir inloggad som 'admin' då frågan '1'='1' uppfylls då den alltid är sann.

### 4.2.4 Remote File Inclusion (RFI)

Angriparens främsta önskan är att kunna köra sin kod på målsystemet; en RFI [23] möjliggör detta. En lyckad RFI ger angriparen åtkomst till PHP-programmets funktioner, så som databaskommunikation, åtkomst till lösenord och filer med mera.

Ett typiskt exempel på en sårbar PHP skript:

```
1 <?php
2     ...
3     include($path . "/file.php");
4     ...
5 ?>
```

Ett giltig begäran till skriptet kan vara:

```
1 http://target.com/index.php?path=file.php
```

En angripare kan då ändra på detta och försöka köra sin egen kod på servern genom att skriva in följande text i adressfältet:

```
1 http://target.com/index.php?path=http://site.com/evilkod.txt?
```

### 4.2.5 Remote Code Execution (RCE)

Remote Code Execution eller RCE [20] är ett säkerhetsproblem i PHP som möjliggör fjärrkörning av kod via filsystemsanrop. Den här typen av säkerhetshål kan ske på grund av olika saker:

- Otillräcklig validering av användarnas input innan anrop av det dynamiska filsystemet
- allow_url_fopen() funktionen är aktiverad i PHPs grundinställningar vilket inte ens används av många webb-applikationer.

Ett typiskt PHP-kod som är sårbart mot RCE:

```
1 <?php
2     $msg = $_GET['msg'];
3     $ip = getenv('REMOTE_ADDR');
4     $error = fopen('errorlog.php','a');
5     fwrite($error,'<br />'.$msg.'<br />'.$ip.'<br />');
6     fclose($error);
7 ?>
```

Det kan utnyttjas av en angripare på detta sätt:

```
1 http://target.com/info.php?msg=<?passthru(&_get('EVILCODE']);?>
```

Angriparen kan då utnyttja sårbarheten genom att anropa på errorlog.php:

```
1 http://target.com/errorlog.php?attacker=http://site.com/evilCode.txt?
```

## 4.3 Webbapplikations-fuzzer

Webbapplikations-fuzzer är en specialiserad form av nätverksprotokoll-fuzzers. Till skillnad från en nätverksprotokoll-fuzzer som muterar alla nätverkspaketstyper fokuserar webbapplikations-fuzzer bara på de paket som överensstämmer med HTTP-specifikationen. Det vill säga det görs ett antal tester mot en applikation som körs på en server. Målet med testerna är att hitta de buggar i webbaplikationer som låter en angripare ta kontroll över servern som webbapplikationen körs på. En *domän*-webbapplikations-fuzzer är en webbapplikations-fuzzer som testar ett flertal applikationer över en domän. Detta förklaras utförligt i nästa kapitel.

# Kapitel 5

# Crawler

En annan viktig del i en webbapplikationsfuzzer är att den ska ha möjlighet att kunna testa ett flertal webbapplikationer över en större domän. Dessa stora domäner är ganska vanliga hos större organisationer. Det är också vanligt att olika delar av en och samma organisation utvecklar olika webbapplikationer för sina respektive behov. Ibland kan olika webbapplikationer kommunicera med varandra och/eller med någon underliggande databas. Det kan därför vara angeläget att köra webbfuzzern mot alla webbapplikationer i en domän/subdomän.

Ett sätt att kunna göra det på är med hjälp av en s.k. “crawler” eller “spider” som de också kallas [27]. En crawler är en applikation som söker igenom en webbsida efter länkar till andra webbsidor, eller till andra delar av samma webbsida. Denna operation sker rekursivt och kan ta allt från några få sekunder till flera timmar eller mer. Hur snabbt det går kan bero på en eller flera av följande faktorer:

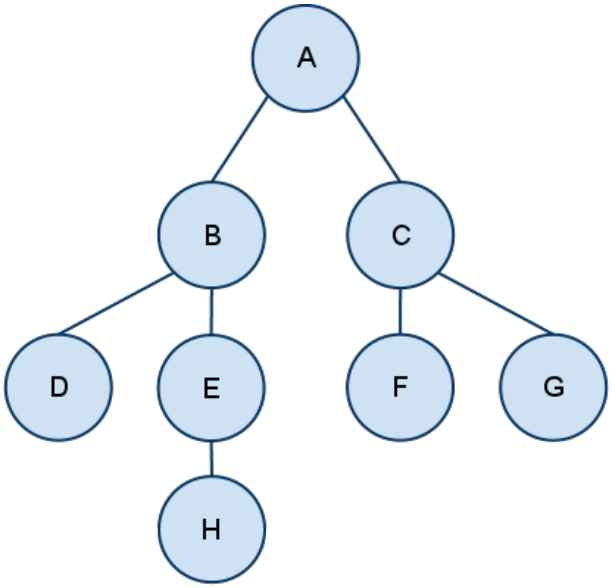
1. hur stor webbsidan är
2. hur brett man vill söka
3. hur effektiv den implementerade algoritmen för crawlern är
4. webbserverns och klientens prestanda
5. webbserverns och klientens nätverkshastighet

Punkterna 1, 4 och 5 kan uteslutas i det här fallet eftersom ändring av webbsidors storlek samt inköp av ny hårdvara ej är aktuell. Det kan dock vara intressant att ha dessa punkter i åtanke om det skulle uppstå fel.

## 5.1 Traversering

Inom datalogi kan man abstrahera ett dator-nätverk eller valfritt sammansatta objekt som ett träd eller graf, och objekten som noder eller förgreningar. I detta fall är varje nod en webbsida och varje förgrening representerar en länk [12].

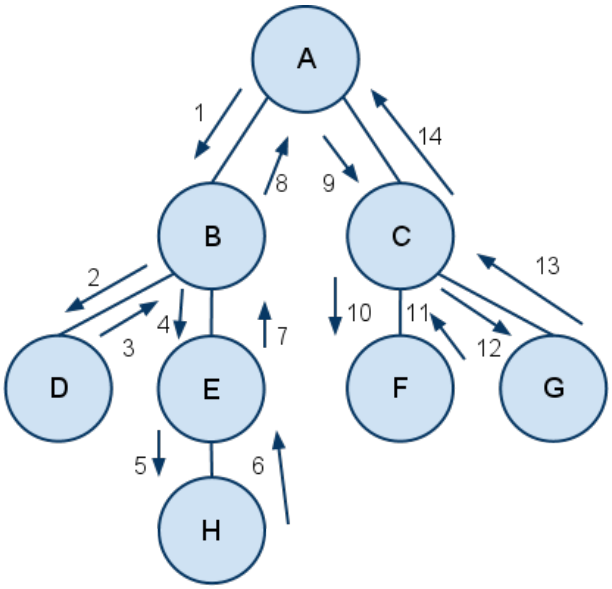
Traversering är en systematisk procedur för att utforska en graf genom att undersöka alla dess noder och förgreningar. Det finns idag två dominerande algoritmer för att styra på vilket sätt traverseringen skall genomföras.




**Figur 5.1.** Ett träd med 8 noder varav en rot

### 5.1.1 DFS

En av dessa två algoritmer är djupet-först-sökning (DFS). DFS är definierad som så att den går in i (besöker) en undernod rekursivt, tills den har nått botten på trädet. När den har nått botten kommer den sedan upp ett steg för att på nytt försöka gå ner till botten via en annan gren. På så sätt kommer den genomsöka alla noder i trädet. En illustration på det visas nedan.




**Figur 5.2.** Illustrering av DFS på ett träd med 8 noder

En traversering av ovanstående träd blir då som följande:
$A \rightarrow B \rightarrow D \rightarrow E \rightarrow H \rightarrow C \rightarrow F \rightarrow G$

En mer generell algoritm för DFS [12, p. 305]:

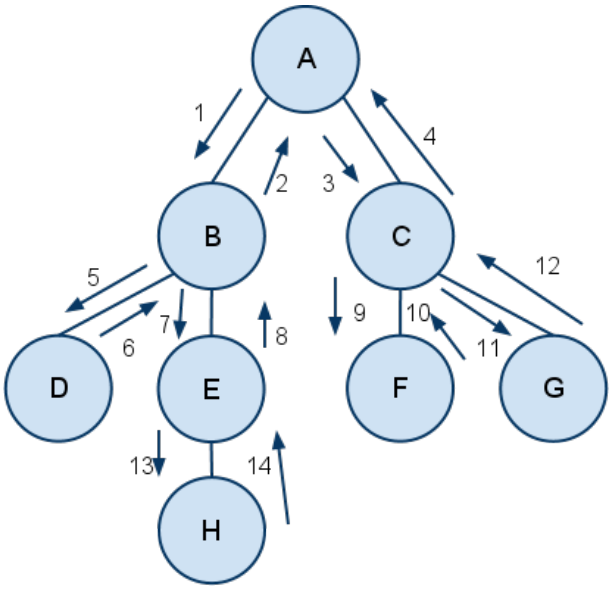
```
1  Algoritm DFS(G, v):
2  Input: A graph G and a vertex v of G
3  Output: A labeling of the edges in the connected component of v as
       discovery edges and black edges
4
5  label v as explored
6  for all edges e in G.incidentEdges(v) do
7      if edge e is unexplored then
8         w <= G.opposite(v, e)
9         if vertex w is unexplored then
10            label e as dicovery edge
11            recursively call DFS(G, w)
12        else
13           label e as a back edge
```

### 5.1.2 BFS

Bredden-först-sökning (BFS) -algoritmen är definierad på följande sätt:

1. Spara först alla närliggande noder i en kö.
2. Besök sedan dessa noder i den ordning de finns i kön och lägg till deras undernoder till kön.

Denna procedur sker iterativt tills det inte finns några noder kvar i kön. Nedan följer en illustration på hur BFS går igenom samma träd som ovan.




**Figur 5.3.** Illustrering av BFS på ett träd med 8 noder

En traversering av ovanstående träd blir då som följande:
$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G \rightarrow H$

En mer generell algoritm för BFS [12, p. 313]:

```
1  Algoritm BFS(G, s):
2  Input: A graph G and a vertex s of G
3  Output: A labeling of the edges in the connected component of v as
       discovery edges and black edges
4
5  create an empty container L0
6  insert s into L0
7  i <= 0
8  while Li is not empty do
9      create an empty container Li+1
10     for each vertex v in Li do
```

```
11          for all edges e in G.incidentEdges(v) do
12              if edges e is unexplored then
13                  let w be the other endpoint of e
14                  if vertex w is unexplored then
15                      label e as discovery edge
16                      insert w into Li+1
17                  else
18                      label e as a cross edge
19      i <= i + 1
```

### 5.1.3 Jämförelse mellan DFS och BFS

Även då BFS och DFS båda har sina för- och nackdelar i olika scenarion så är det nästan ingen skillnad alls i sådana här fall då en crawler måste gå igenom hela hemsidan. Dock kan det fortfarande vara intressant att implementera båda två om det är av intresse att traversera en domän i en viss ordning.

## 5.2 Identifiering av gilltiga länkar

För att kunna hitta de sökta länkarna i en HTML-kod, måste man ta reda på exakt vad det är som man söker och därefter generalisera en sökningsmetod för det. I det här fallet handlar det om länkar till webbapplikationer och andra sidor inom samma domän. Nedan listas några exemplar:

```
href = "http://dir/page.jsp?name1=value1"
href = "https://dir/page.asp?name1=value1"
href = "http://dir/page.php?name1=value1&name2=value2"
href = "http://dir/?name1=value1"
href = "/page.htm"
href = "/page.html"
href = "../page.html"
href = "../../page.htm"
```

Det finns många fler exemplar som kan listas här ovan, men detta räcker nog för att generalisera en söknings-/identifierings-metod. En generalisering kan se ut på följande sätt:

```
[HTML_tag][protokoll]://[sökväg]/[sida][.filändelse]?[namn]=[data](&[namn]=[data])+
```

`[HTML_tag]`- innehåller i detta fall endast "href="
`[protokoll]` - kan innehålla både “http” och “https”, men får också vara tom.
`[sökväg]` - måste alltid börja med en eller flera gånger av någon av följande: `'/', '../'`. Sedan följer en eller flera tecken som inte är någon av följande: ’?’, ’=’.
`[sida]` - en sida är precis som ovan, fast utan det första kriteriet.
`[filändelse]` - här är det någon av följande som gäller. .asp, .aspx, .php, .jsp och .cgi. Det kan också vara tomt.

`[namn]` - precis som [sida]
`[data]` - precis som [sida]
`(&[namn]=[data])` - om webbapplikationen tar flera argument som indata så separeras de med ett '&'-tecken. Därefter följer `"[namn]=[data]"` för varje argument.

Det absolut vanligaste sättet att särskilja liknande sammansatta strängar ur en större text är att använda ett s.k. reguljär uttryck. Detta kan göras i nästan vilken programmeringsspråk som helst, och på många olika sätt. Reguljära uttryck är dock långt ifrån det enda sättet att t.ex. hitta länkar i en webbsida. T.ex. webbläsarna själva använder inte reguljära uttryck för att hitta länkar. Det finns t.ex. HTML-parsningbibliotek såsom "tagsoup" och "beautiful soup" som "plockar ut" länkar ur ett dokument med DOM-innehåll (HTML, XML och dylikt).

# Kapitel 6

# Verktyg

Det finns ett otal verktyg till förfogande för att kunna skapa en fuzzer eller analysera ett protokoll. Den absolut enklaste formen av en fuzzer kan man enkelt skapa genom att utnyttja pipe-funktionen i unix-liknande miljö tillsammans med ett antal kommando i terminalen. Det är också ganska vanligt att skapa ett GUI-baserad program, där man kan ha en bra överblick av den grafisk representation av det som avses fuzzas. Några kommersiella program som hör till den senare kategorin är *WebInspect* [7], *Rational App Scan* [8], *Acunetix Web Vulnerability Scanner* [1] med flera.

Andra verktyg som kan vara av intresse för den här rapporten är bl.a. LiveHttpHeader [3], som är ett tillägg till webbläsaren Firefox. Den fungerar som en enkel sniffer som lyssnar till vilka HTML-headers som skickas och tas emot i Firefox. Även välkända WireShark [28] kan komma till bra användning för att sniffa på nätverkstrafiken, för att sedan kunna analysera in- och ut-datan.

Det finns också en uppsjö av olika fuzzingramverk tillgängligt som kan förenkla utvecklingen av fuzzer-applikationer avsevärt. Det brukar dock finnas en inlärningströskel som kan försvåra valet. Bland de mest populära fuzzer-ramverk är: *Spike* [9], *Peach* [15], *Codenomicon* [5], *Sulley* [6], *JBroFuzz* [14], *Autodafé* [19], *General Purpose Fuzzer* [10] och *Burp Sutie* [16].

# Kapitel 7

# Implementation

Det här kapitlet förklarar hur implementationen av Domän-Webb-Applikations-Fuzzer (DWAF) har gått till. En avskalning av vissa funktioner har gjorts för att hålla fuzzern på en relativ enkel nivå och p.g.a. tidsbrist. Fokus har lagts på att få fuzzern att framgångsrikt kunna exploatera två sårbarhetstyper mot webbapplikationer. De två sårbarhetstyperna är SQL-injection och XSS. Vidare är en crawler implementerad med både DFS och BFS algoritm. Den senare är flertrådad och avslutar därför arbetet mycket fortare än den tidigare. En mer detaljerad förklaring följer nedan.

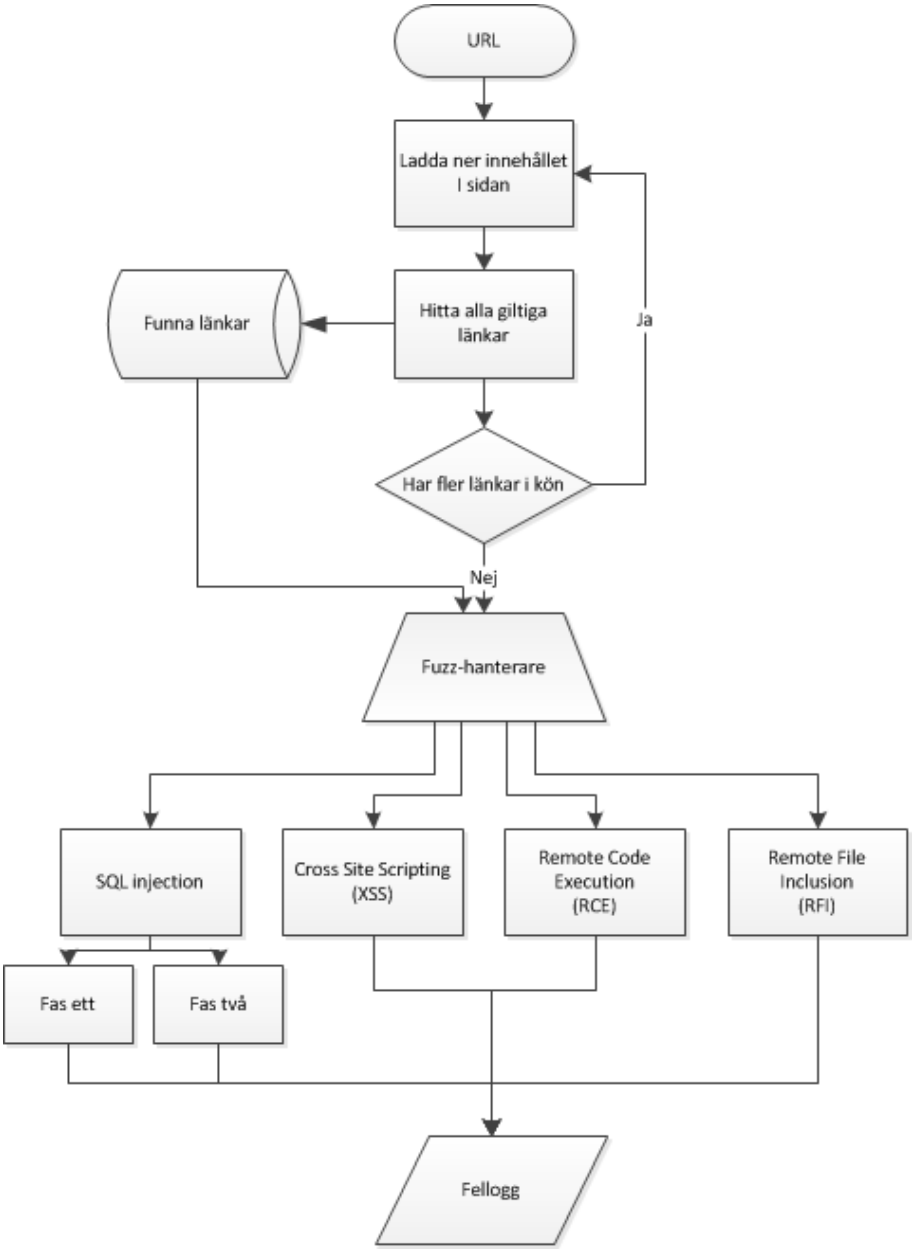
## 7.1 Var ska man börja?

För att kunna implementera en fuzzer, måste hänsyn tas till följande sex punkter:

1. **Identifiera målet:** I det här fallet är det alla webbapplikationer som tar indata av typen som är beskrivet nedan.
2. **Identifiera indatan:** Indatan ska vara riktade mot alla funna länkar i en domän som tar indata av typen som följs av ett ’?’ och sedan en eller flera av “namn=värde”-typen, med ett ’&’ emellan om det är flera.
3. **Generera fuzzad data:** Då fokus har lagts på sårbarhetstyperna SQL-injection och XSS, är det enklast att hårdkoda de indata som är välkända till att exploatera buggar eller felmeddelanden.
4. **Exekvera fuzzad data:** Exekvering av fuzzad data sker genom att det genererade fuzz-data läggs till i slutet av en av en länk och sedan utförs ett anslutningsförsök.
5. **Observera undantagen (fr. eng. exceptions):** Efter en lyckad anslutning, skickas utdatan till en felloggshanterare.
6. **Avgöra exploateringen/buggidentifieringen:** I felloggshanteraren analyseras utdatan och identifierar funna buggar och/eller felmeddelanden.

## 7.2 Pseudokod och diagram

Det här avsnittet avser att beskriva de viktiga psudokoderna och DWAFs UML-diagram.




**Figur 7.1.** UML-diagram för implementation av en web-fuzzer.

Ovanstående UML beskriver hur DWAF arbetar. Den är i huvudsak uppbyggd av två delar. En crawler-del och en fuzzer-del. Först får crawlern en länk (URL) som den söker

igenom. Den samlar då alla funna giltiga länkar i en lista. Sedan går den igenom alla de funna länkarna och lägger till dem i listan/kön. På så sätt hittar den alla länkar i en domän. Länkarna identifieras med hjälp av en reguljär uttryck som börjar med "href" och som följs därefter av den sökta länken:

```
1 href ?= ?(?:\"|')((http[s]?://)?([\\w+\\x2e/?]+)?(/?(\\w+)?((.asp|.aspx
    |.php|.jsp|.cgi)\\?((\\w+=\\w+&?)+)))?)(\"|')}}
```

Regexpet ovan är konstruerat så att det grupperar olika delar av länken för att möjliggöra ett smidigt åtkommande. Till exempel kan man i Java anropa metoden `match.group(0)` för att få tillgång till hela strängen, och `match.group(1)` för att komma åt första grupperingen som i det här fallet blir `http[s]://`.

### 7.2.1 Algoritm

Även om crawler-algoritmen är av stor betydelse så är det ändå relativ enkelt att implementera en naiv sådan. Pseudokoden (för en DFS variant) ser ut som följande:

```
1  function crawl_DFS is:
2  input: String url
3  external: HashSet<String> visited
4      siteContent <= download(url)
5      validLinks  <= getValidLinks(siteContent, url)
6      for all urls in validLinks
7              if urls not visited
8                     visited.add(urls)
9                     crawl_DFS(urls)
10             end if
11     end for
12 end crawl_DFS
```

Den rekursiva funktionen crawl_DFS fungerar som så att den får en initial-url som indata. Den laddar sedan ner hela html koden som finns i den sidan och skickar innehållet som indata till funktionen getValidLinks, tillsammans med den angivna url-en. Som returvärde får den en lista med alla url som finns i samma domän. Sedan anropas crawlern på varje url som finns i listan och som inte redan är besökt. På så sätt går den igenom en hel domän.

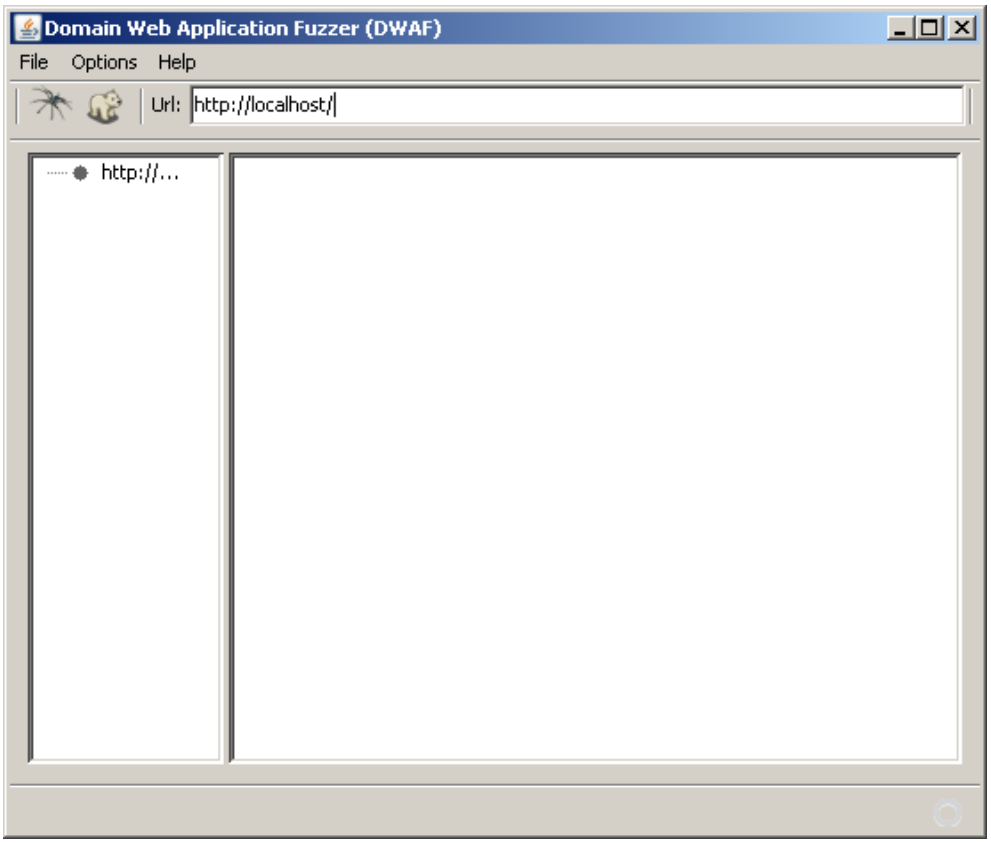
Algoritmen kan effektiviseras genom att göra sökningen flertrådad så att flera sidor laddas ner och analyseras samtidigt.

```
1 function getValidLinks is:
2 input: String siteContent, String url
3 output: ArrayList<String>  validLinks
4 external: ArrayList<String> linkToFuzz
5     for all urls in siteContent
6           if urls matches HTTP_PATTERN
```

```
7                   validLinks.add(urls)
8               end if
9               if urls matches FUZZ_PATTERN and starts with url
10                  linkToFuzz.add(urls)
11              end if
12      end for
13  end getValidLinks
```

Ovanstående funktion, getValidLinks, anropas från crawler-funktionen. Funktionens uppgift är att gå igenom hela HTML-koden och samla ihop de url som länkar till andra sidor inom samma domän. Den ska också samla ihop alla url på vilka fuzzing kan tillämpas på för senare användning. Här är det viktigt att lägga krut på hur själva identifieringen av länkar ska gå till, vilket har förklarats utförligt i tidigare kapitel.

Efter insamlingen av alla länkar, utförs själva fuzzingen som går igenom alla länkar och utför de olika sårbarhetstesterna parallellt för varje länk och för varje sårbarhetstyp.



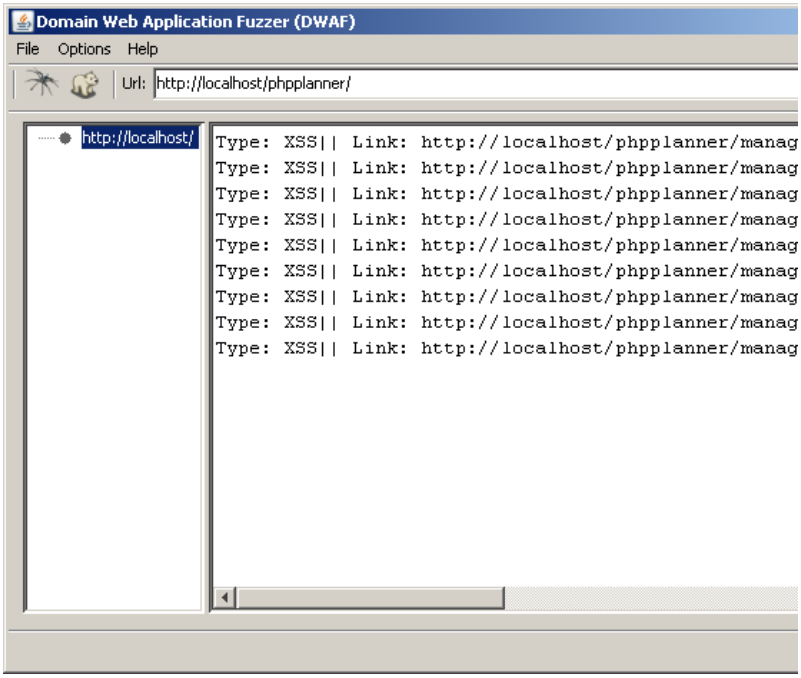

**Figur 7.2.** En första prototyp av DWAF.

# Kapitel 8

# Resultat

Rapportens ursprungliga uppgift var att redogöra vad en fuzzer är, dess olika funktioner, varför fuzzer-testning är viktigt och hur man implementerar en sådan. Nedan visas de resultat som är framtagna efter lyckad fuzzing av olika webbapplikationer med DWAF.

*PHP Planner* [18] är en enkel webbapplikations-kalender med ett fullt fungerande användarsystem med möjlighet att anpassa det allmänna utseendet och beteendet i kalendern. Efter installation av PHP Planner på localhost, fuzzades den med DWAF och det visade sig att applikationen är sårbar mot XSS.

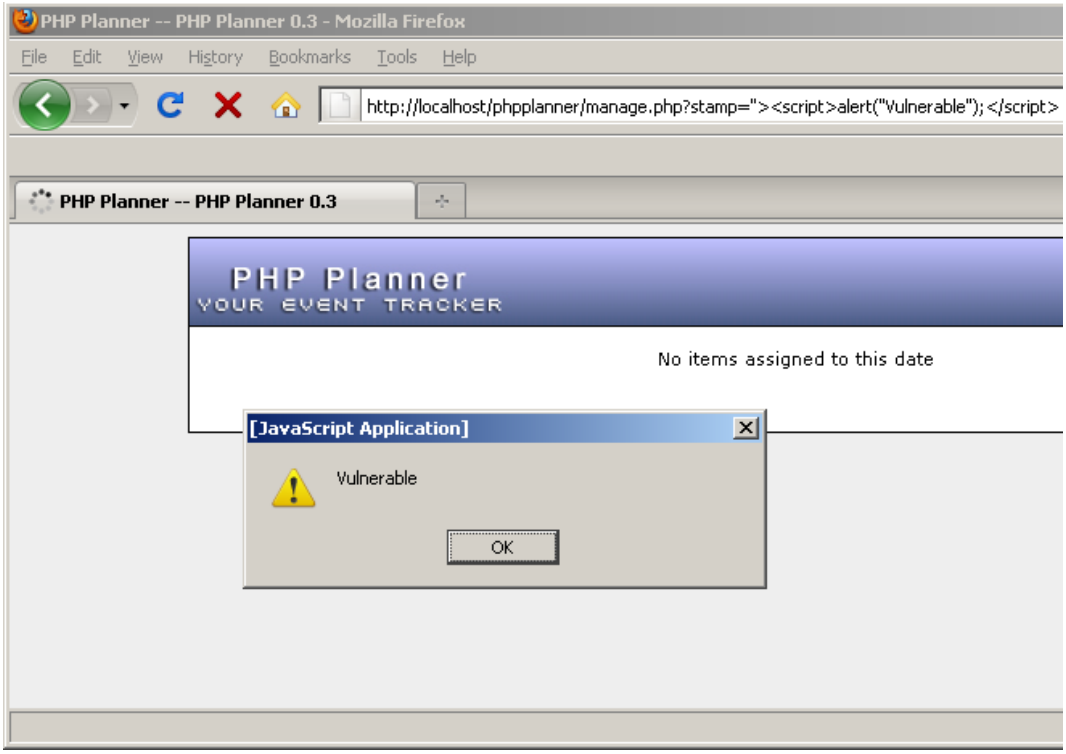



**Figur 8.1.** Funna XSS i PHP Planner med DWAF.

För att kontrollera om resultatet från DWAF var korrekt testades det manuellt i webb-läsaren. I figuren nedan bekräftas DWAFs resultat. Det som sker är att webbapplikationen möjliggör att en användare kan köra valfri HTML-kod direkt från adressfältet. Koden som körs i webb-läsaren är:

```
1 http://localhost/phpplanner/manage.php?stamp="><script>alert("Vulnerable
    ");</script>
```

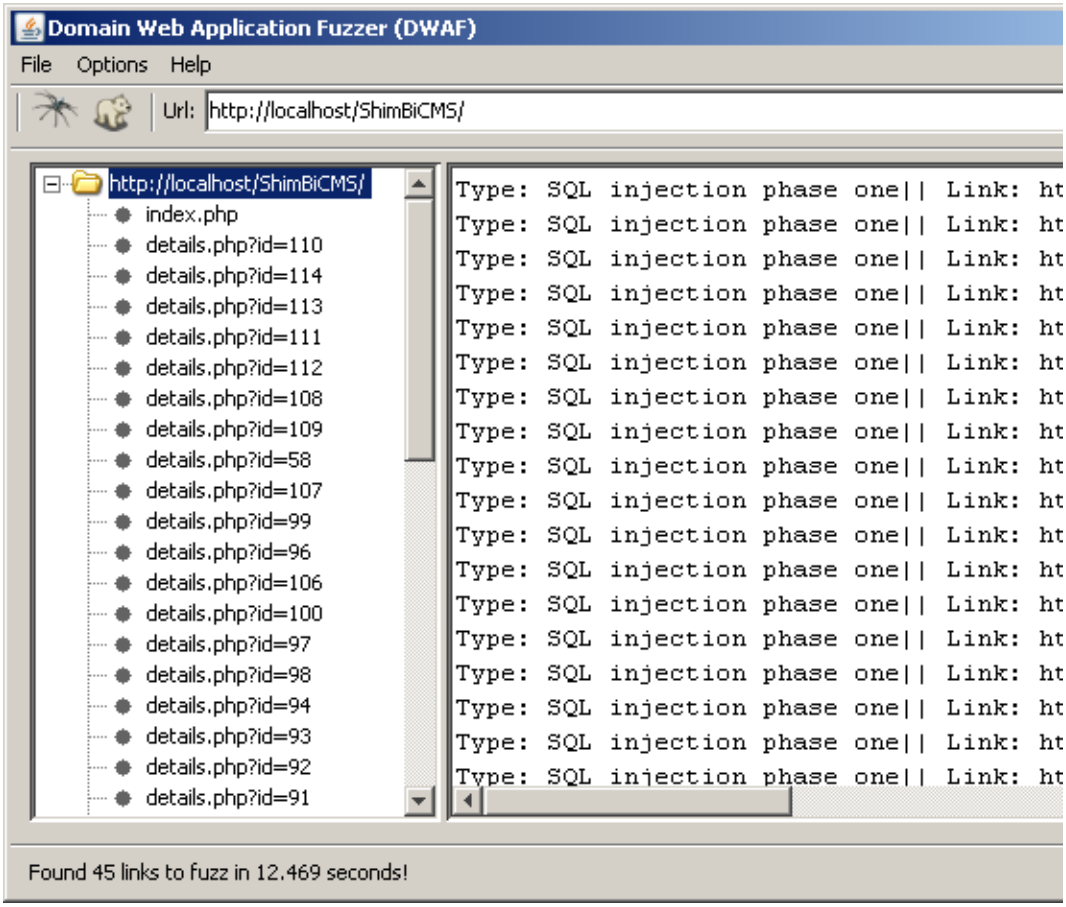
Detta är en enkel Javaskript som anger ordet “Vulnerable” som meddelande på skärmen.




**Figur 8.2.** Manuellt test av XSS i PHP Planner.

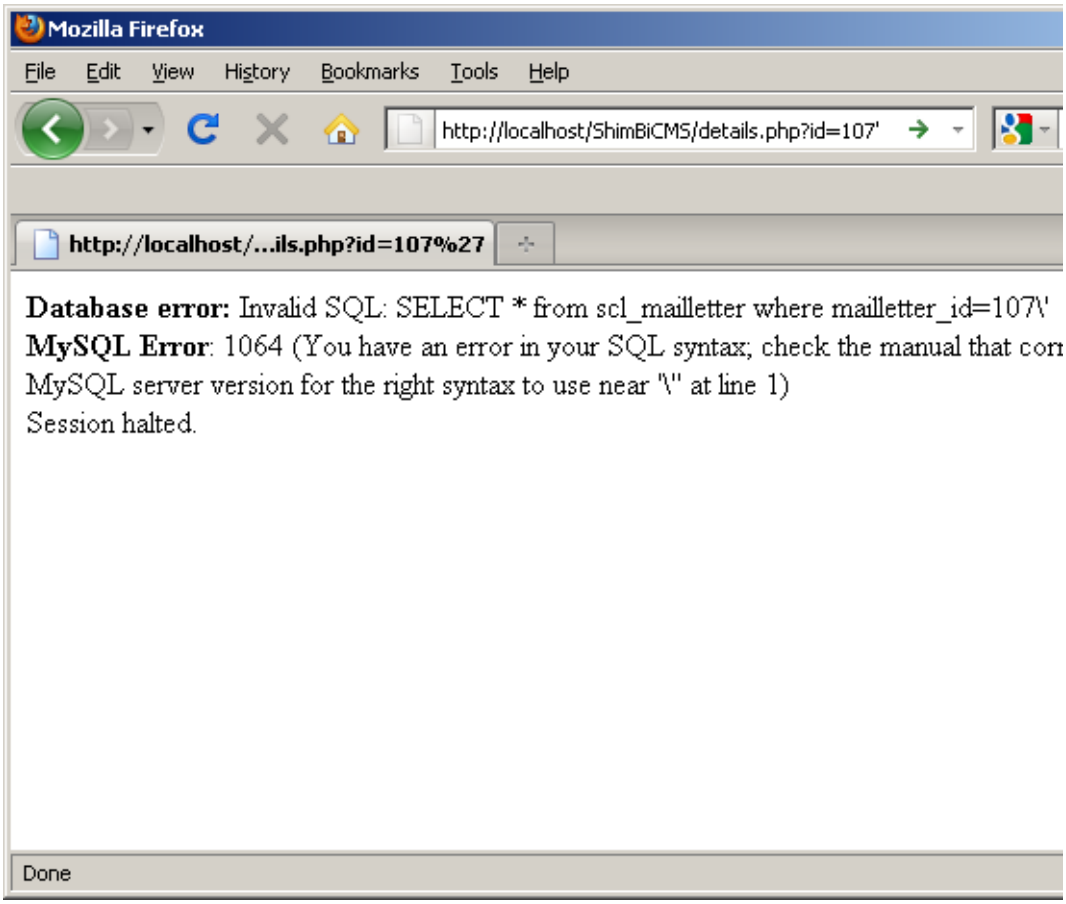
*ShimBi CMS* [17] är en webbapplikation som används för att kunna skapa hemsidor och lätt kunna hantera dem. ShimBi CMS installerades på localhost servern och fuzzades, efter ett tag visade det sig att applikationen skall vara sårbar mot SQL-injection. DWAF visar typen på buggen är av typen *pahse one* vilket innebär att efter en inmatning av testdata, genererar php ett felmeddelande vilket skickas direkt till klienten.

Om DWAF visar typen *phase two* då innebär det att inget felmeddelande genereras utan angriparen kan mata in villkorliga SQL-kommandon i URLn och se till att resultatet på villkorliga satser skickas till databasen och resultatet skickas till angriparen.



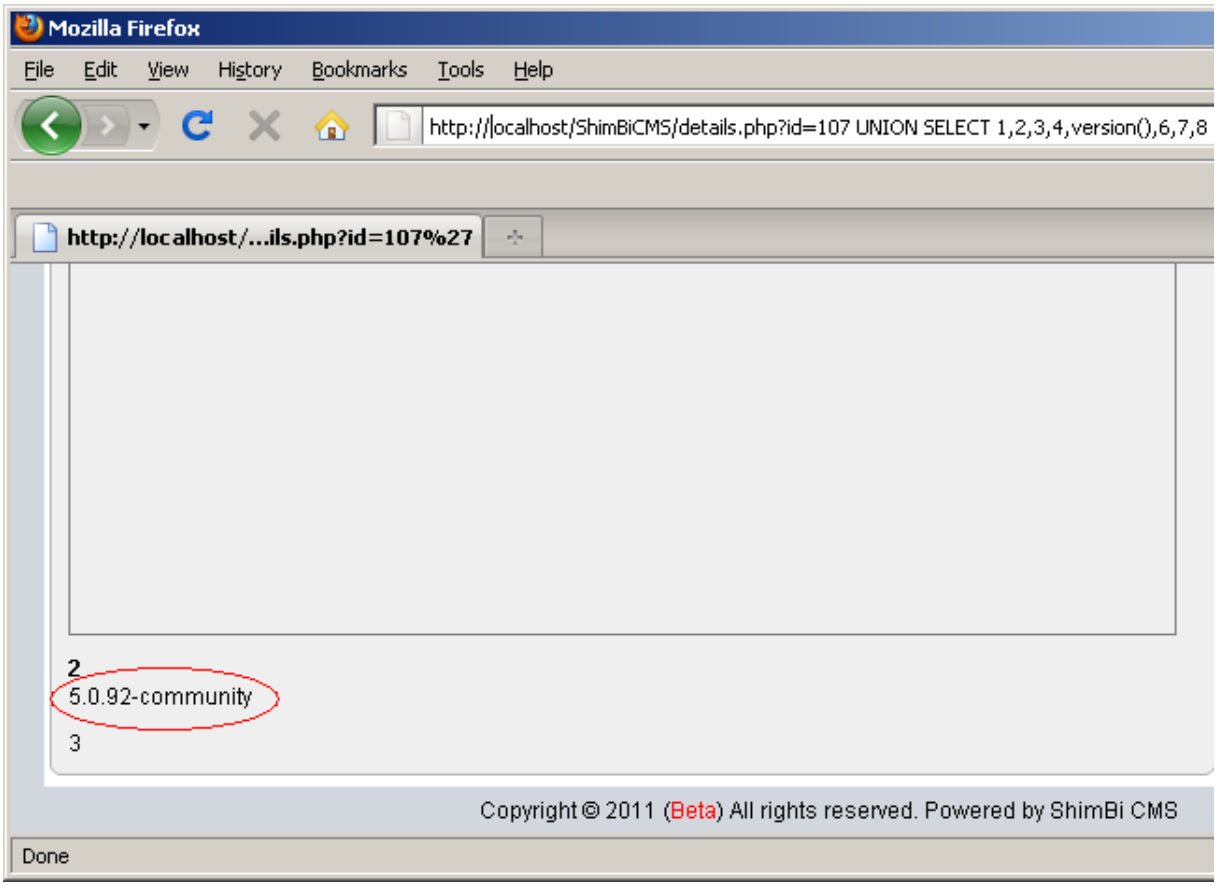

**Figur 8.3.** Funna SQL-injection i *ShimBi CMS* med DWAF.

Även här testades resultatet från DWAF manuellt för att bekräfta dess tillförlitlighet. I nedanstående figur illustreras hur ett SQL-kommando körs direkt från webbläsarens adressfält. Det visar tydligt på att webbapplikationen inte hanterar inmatningar av den här typen och skickar därför hela SQL-kommandot direkt till SQL-databasen. Därefter skickar databasen tillbaka ett felmeddelande.




**Figur 8.4.** Manuellt test av SQL-injection i *ShimBi CMS*.

Nedan visas ett test där man försöker ta reda på versionen på den installerade MySql-applikationen på servern. Genom att ändra på SQL kommandot kan man även ta reda på värdefull information så som användarnamn och lösenord. På så sätt kan man ta kontroll över servern.




**Figur 8.5.** Manuellt test av SQL-injection i *ShimBi CMS*.

# Kapitel 9

# Diskussion

Av resultaten i föregående avsnitt ser vi tydlig en enkel men väl fungerande webbapplikationsfuzzer som identifierar säkerhetsbuggar av typen SQL-injection och XSS över en domän med flera webbapplikationer. Det går även att med små modifieringar lägga till andra sorters sårbarhetstyper av typen LFI, RFI. RCE och möjligtvis även andra sårbarhetstyper som använder modifiering av en URL för att hitta buggar och säkerhetshål.

Det måste dock tilläggas att ett av huvudsyftena med det här projektet var att efter en lyckad implementation av fuzzer-applikationen så skulle CSCs rapp-system testas[1]. Detta var under planeringsfasen lite underskattat och resulterade i att vi stötte på problem som skulle tvinga oss att ändra stora delar av en välfungerande kod. Problemet låg i att för att kunna fuzza olika delar av Rapp så måste en inloggning ske först. Inloggningen sker med hjälp av kakor som Rapp tillhandahåller från inloggning på ett kth.se-konto. Då inloggning på nätet sker på olika sätt och p.g.a. tidsbrist valde vi att inte göra några större förändringar. Vi kan därför dra den slutsatsen att det är ganska svårt att göra generella fuzzers som också är lätta att modifiera och använda.

Crawler-delen var till en början ganska lätt att implementera. Men vid identifiering av korrekta länkar blev det ganska komplicerad och tidskrävande. Här skulle ett väletablerat API, parser [13], DOM eller en crawler-applikation vara att föredra. Vi såg också att det var stor skillnad mellan att traversera enkeltrådad och att traversera flertrådad. Det gick närmare bestämd 5-10 gånger snabbare med en flertrådad traverser.

Buggidentifiering (felloggen) i DWAF fungerar ganska bra. Det som dock har varit svårt att implementera är att avgöra när det blir s.k. “false-positive”. Detta sker ganska ofta i fas-2 (phase two) av SQL-injection testet. Detta sker pga. att sidan som ska fuzzas måste laddas först en gång och spara innehållet, och sedan matas samma sida med fuzzdata och sidan laddas om på nytt. Sedan sker en jämförelse mellan dessa två omgångar. Om

[1]Rapp är ett förhållandevis nytt resultatrapporteringssystem för CSCs kursansvariga och kursdeltagare. Rapp är intressant för webapplikations-fuzzning för att det är ett relativt nytt system, och därför är ett rimligt antagande att många buggar ännu inte upptäckts [2].

innehållet i sidorna skiljer sig från varandra, så registreras det som en bugg. Problemet är att vid de tillfällen där sidorna innehåller inbäddad dynamisk reklam, eller annan dynamisk data så kan en “false-positive”-bugg registreras. Här skulle man kunna använda en befintlig eller implementera någon sorts bättre jämförelsemetod än den vanliga tecken-mot-tecken jämförelse som finns implementerad i java biblioteket.

# Litteraturförteckning


[1] Acunetix. Acunetix web vulnerability scanner. http://www.acunetix.com/vulnerability-scanner/, April 2011.

[2] CSC. Rapp. https://rapp.nada.kth.se/rapp/, Juni 2011.

[3] Nikolas Coukouma Daniel Savard. Livehttpheaders - firefox addon. http://livehttpheaders.mozdev.org/, April 2011.

[4] Exploit DB. Local file inclusion. www.exploit-db.com/download_pdf/13678/, April 2011.

[5] Codenomicon defensics. Codenomicon. http://www.codenomicon.com/, Juni 2011.

[6] fuzzing.org. Fuzzing.org. http://www.fuzzing.org/, Juni 2011.

[7] HP. Webinspect. https://www.fortify.com/products/web_inspect.html, April 2011.

[8] IBM. Rational appscan. http://www-01.ibm.com/software/awdtools/appscan/, April 2011.

[9] Immunity. Spike. http://www.immunitysec.com/resources-freesoftware.shtml, Juni 2011.

[10] VDA Labs. General purpose fuzzer. http://www.vdalabs.com/tools/efs_gpf.html, Juni 2011.

[11] Pedram Amini Michael Sutton, Adam Greene. *Fuzzing - Brute Force Vulnerability Discovery*. 2007.

[12] Roberto Tamassia Michael T. Goodrich. *Algorithm design*. 2002.

[13] OWASP. Antiparser. http://antiparser.sourceforge.net/, April 2011.

[14] OWASP. Jbrofuzz. https://www.owasp.org/index.php/JBroFuzz, Juni 2011.

[15] Peach Fuzzing Platform. Peach. http://peachfuzzer.com/, Juni 2011.

[16] PortSwigger. Burp suite. http://portswigger.net/burp/, Juni 2011.

[17] ShimBi. Shimbi cms. http://cmsen2.shimbi.in/, April 2011.

[18] Tom Sommer. Php planner. http://phpplanner.sourceforge.net/, April 2011.

[19] Martin Vuagnoux. Autodafé. http://autodafe.sourceforge.net/, Juni 2011.

[20] Wikipedia. Arbitrary code execution, also known as remote code execution. http://en.wikipedia.org/wiki/Arbitrary_code_execution, April 2011.

[21] Wikipedia. Cross-site scripting (xss). http://en.wikipedia.org/wiki/Cross-site_scripting, April 2011.

[22] Wikipedia. Fuzz testing. http://en.wikipedia.org/wiki/Fuzz_testing, April 2011.

[23] Wikipedia. Remote file inclusion (rfi). http://en.wikipedia.org/wiki/Remote_file_inclusion, April 2011.

[24] Wikipedia. Sql-injection. http://en.wikipedia.org/wiki/SQL_injection, April 2011.

[25] Wikipedia. System testing. http://en.wikipedia.org/wiki/System_testing, Juni 2011.

[26] Wikipedia. Unit testing. http://en.wikipedia.org/wiki/Unit_testing, Juni 2011.

[27] Wikipedia. Web crawler. http://en.wikipedia.org/wiki/Web_crawler, April 2011.

[28] Wikipedia. Wireshark. http://en.wikipedia.org/wiki/Wireshark, April 2011.

# Bilaga A

# Länk till DWAF-material

Följande länk är till fuzzer-programmet DWAF inklusive dess källkod. Det är fritt fram att ladda ner, ändra och vidaredistribuera under GPLv3-licens: `http://bitbucket.org/HFM/dwaf`