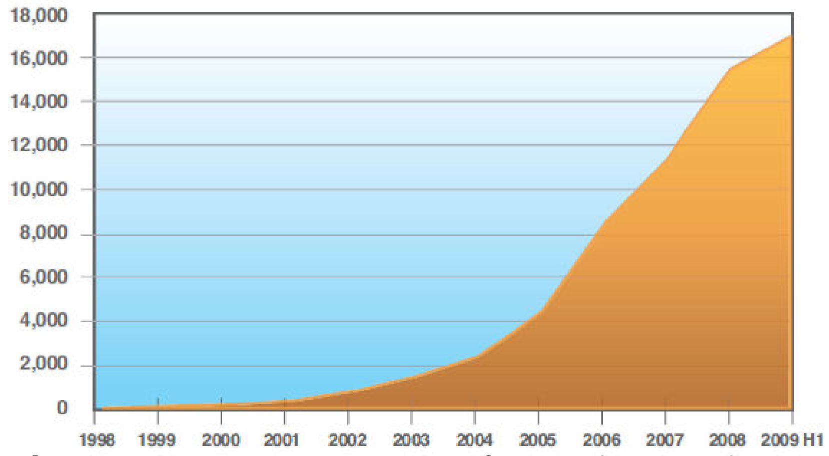
Security Testing of Web Applications: a Search Based Approach for XSS Vulnerabilities

Andrea Avancini, Mariano Ceccato
FBK-Irst, Trento, Italy
{anavancini, ceccato}@fbk.eu

Background

Vulnerability Disclosures Affecting Web applications



[IBM Internet Security Systems™ X-Force® 2009 Mid-Year Trend and Risk Report]

Web applications are critical in many activities

Security of Web apps is also critical

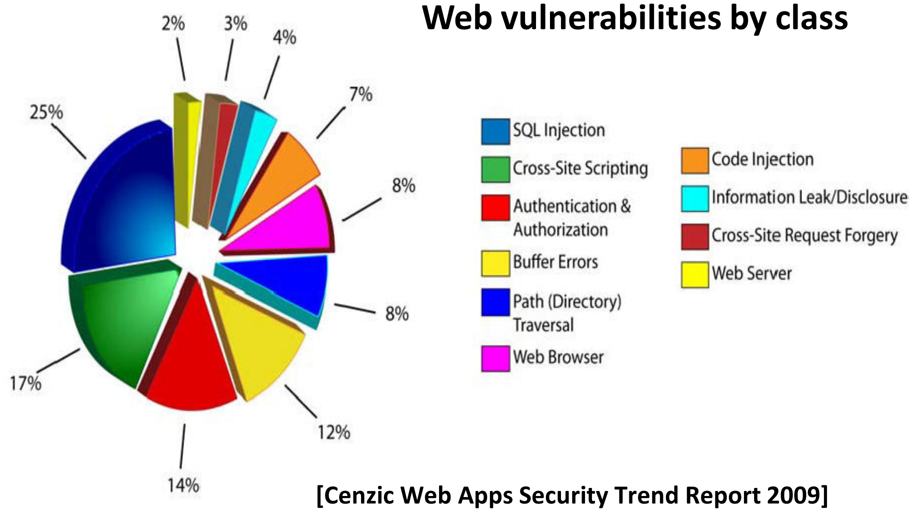
Number of total vulnerabilities in web applications is getting higher year by year

One of the most prominent vuln class is: **Cross-Site Scripting (XSS)**

XSS causes the attacker to inject malicious code into the victim's browser

These vulnerabilities are due to missing or inadequate user input validation

Web vulnerabilities by class

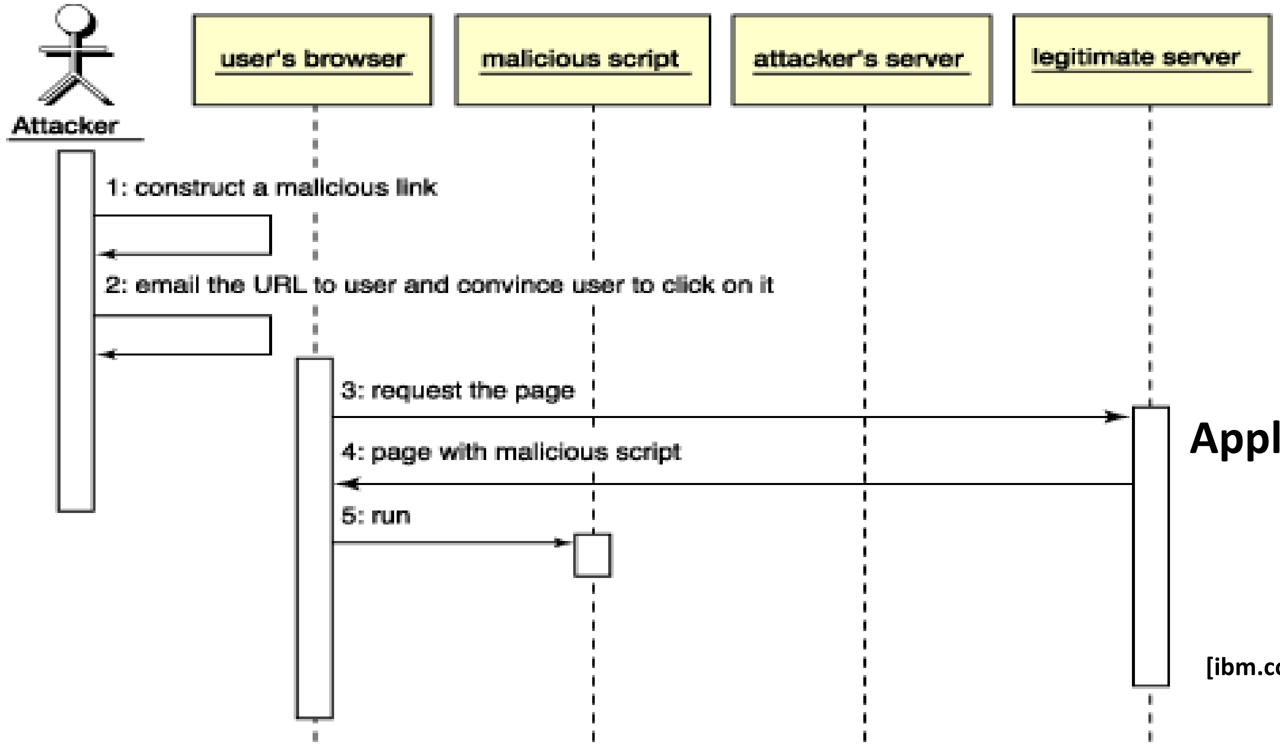


[Cenzic Web Apps Security Trend Report 2009]

Background (II)

Attacker sets the trap – craft malicious link

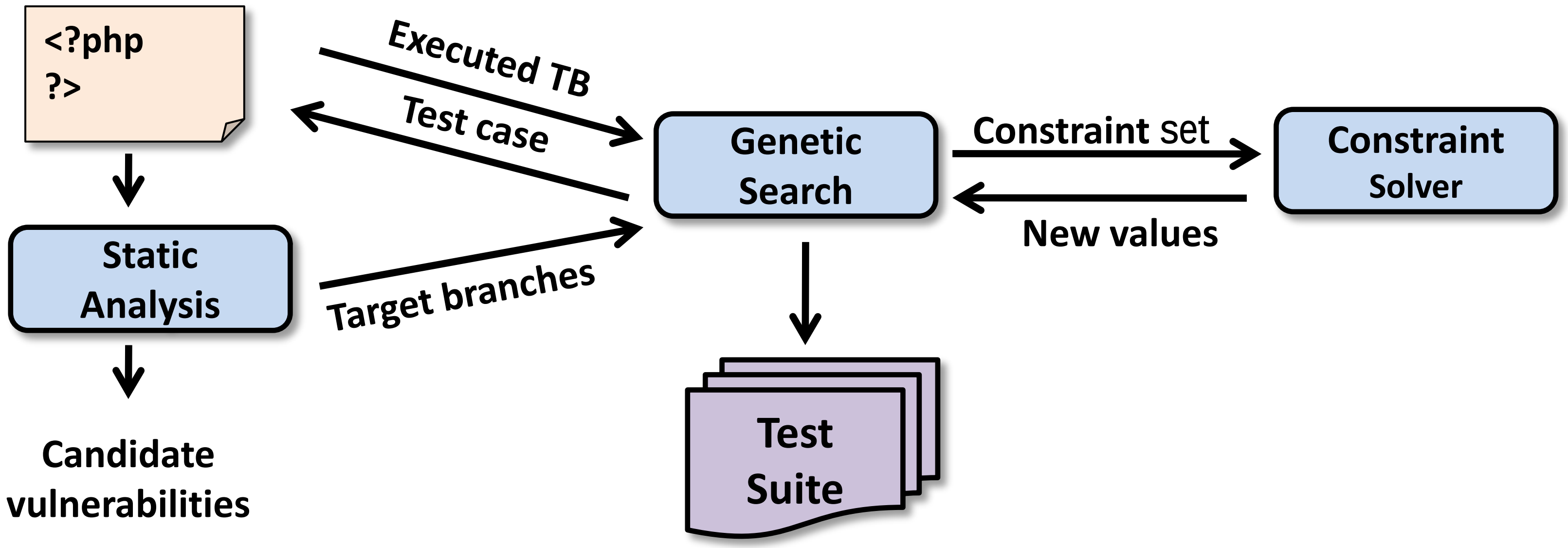
<A HREF=http://legitimateSite.com/registration.cgi?clientprofile=<SCRIPT>malicious code</SCRIPT>>Click here</A>


Attacker
user's browser
malicious script
attacker's server
legitimate server
1: construct a malicious link
2: email the URL to user and convince user to click on it
3: request the page
4: page with malicious script
5: run

Application with reflected XSS vulnerability

[ibm.com]

Our Proposal



Three macro-blocks:

- **Static analysis** (vulnerabilities, target branches)
- **Genetic algorithm** (Global search: test case generation)
- **Constraint solver** (Local search: test case refinement)

Running Example

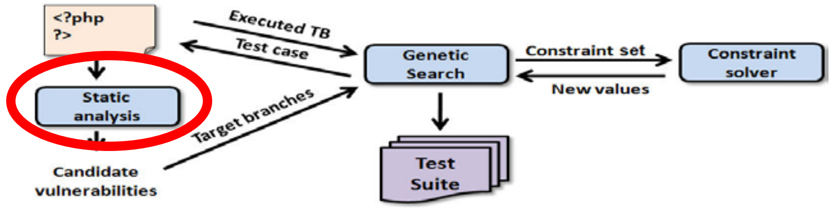
```
1     $user = $_GET[ ”username” ] ;
2     $pass = $_GET[ ”password” ] ;
3     $pass2 = $_GET[ ”password2” ] ;
4     i f ( strpos ( $user , ”<script” ) )
5           $user = htmlspecialchars ( $user ) ;
6     i f ( $user  in  $users )
7           echo ” username  already  taken ” ;
      e l s e
8           i f ( strlen ( $pass )  <  5 )
9                 echo ” password  too  short ” ;
            e l s e
10                i f ( $pass == $pass2 )
11                      new user ( $user , $pass ) ;
12                      echo ”new  account  for ” ;
13                      echo $user ;   / / s i n k
                  e l s e
14                      echo ” passwords  do  not  match ” ;
```

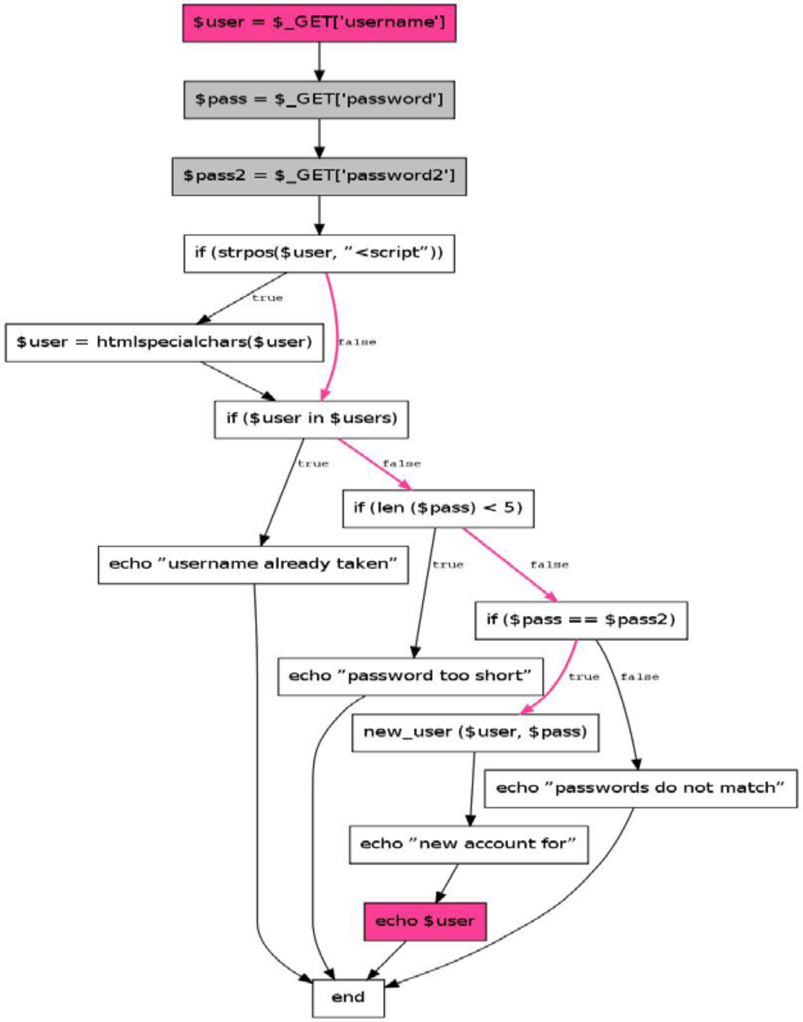
Tainted

Sanitization

Sink

Static Analysis





$user = $_GET['username']
$pass = $_GET['password']
$pass2 = $_GET['password2']
if (strpos($user, "<script"))
true
$user = htmlspecialchars($user)
false
if ($user in $users)
true
false
if (len ($pass) < 5)
echo "username already taken"
true
false
if ($pass == $pass2)
echo "password too short"
true
false
new_user ($user, $pass)
echo "passwords do not match"
echo "new account for"
echo $user
end

Taint Analysis returns

{$user@1, $user@13}

as **assignment chain**

This means that

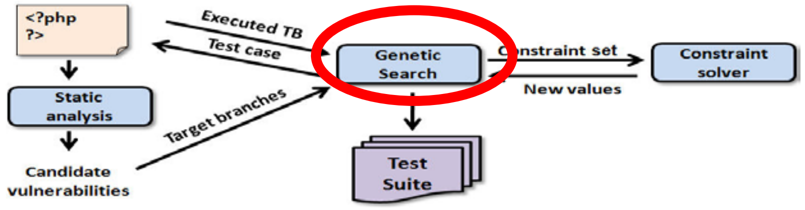
- there exists an input vector that flows to a sink without being sanitized (i.e. there is a vulnerability)
- Statements 1, 13 must be executed to trigger the vulnerability

On top of this information, we calculate control dependencies of the statements in the AC

{4-6, 6-8, 8-10, 10-11}

as **target branches** to execute

Genetic Algorithm



```
population = generateRandomPopulation ( ) ;
for (T  in  vulnerabilities ) {
    while ( not  covered (T)  AND  attempt < maxTry ) {
        selection = select ( population ) ;
        offspring = crossOver ( selection ) ;
        population = mutate ( offspring ) ;
        attempt = attempt + 1 ;
}
```

Fitness function is **approach level**:

– FV(i) = AL(i) = # of target branches executed by individual i

Crossover

{(user, john), (password, xsdsd), (password2, xxxxxx)}

{(user, mark), (password, dfsfa), (password2, zzzzzz)}

{(user, john), (password, xsdsd), (password2, zzzzzz)}

{(user, mark), (password, dfsfa), (password2, xxxxxx)}

Mutation

Add pair

Remove pair

{(user, mark), (password, dfsfa)}

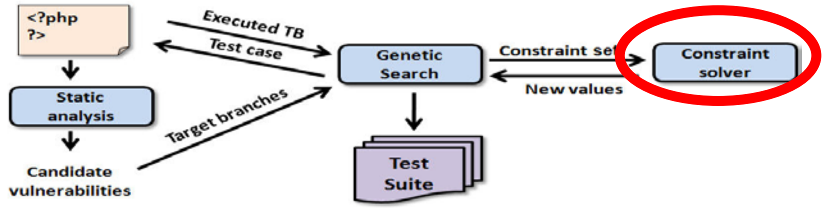
{(user, mark), (password, dfsfa), (password2, xxxxxx)}

{(user, mark), (password, dfsfa), (password2, xxxxxxyyy)}

Change parameter value

{(user, mark), (password, dfsfa), (password2, xxxxTx)}

Constraint Solving



When GA is not able to find a solution, a **constraint solver** is resorted

i = {(user, “ddeerer”), (password, “xxsdsed”), (password2, “dded33e”)}

Branch	Condition	Target Branch
4-6	! strpos(GETusername, ”< script”)	4-6
6-8	false	6-8
8-10	! strlen(GETuser) < 5	8-10
10-14	! GETpassword == GETpassword2	**10-11**

Diverging point is calculated (branch 10-14) and respective constraint is negated

! strpos(GETusername, ”<script”) AND ! strlen(GETusername) < 5 AND GETpassword == GET password2

is passed to solver which could generate:

i_1 = {(user, “ddeerer”), (password, “dsfnggg”), (password2, “dsfnggg”)}

Empirical Results

Run the tool on a real world application

Case study:Yapig 0.95-b

- Open source image gallery app, 53 files, 9 kloc

Page	#	Target Branches Covered			
		0%	38-46%	50-75%	100%
add_comment	1			1	
add_gallery	6		4	1	1
admin	1	1			
delete_gallery	4	2		1	1
modify_gallery	6	3		1	2
modify_phid	6	3		1	2
Slideshow	9				9
Upload	3		2		1
View	2	1		1	
Total	**38**	**10**	**6**	**6**	**16**

Cardinality	#	Target Branches Covered			
		0%	38-46%	50-75%	100%
1-2	15	7		3	5
3	7	2			5
4	7	1		2	4
5-13	9		6	1	2
Total	**38**	**10**	**6**	**6**	**16**

Advantages

- Static analysis is over-conservative
  - No false negatives
- Search space is usually very large but GA heuristic helps in reducing it (global search)
- With the reduced search space, resorting to a constraint solver does not create scalability issues (local search)
- Actual executable test cases are generated for web applications

Limitations

- Static analysis is over-conservative
  - False positives
- GA does not always converge to a solution
- Constraint solving is limited by the use of concrete values when:
  - Symbolic value is not always available or
  - Expressiveness of solver is limited
- Generated test cases are not actual attacks, they do not try to inject malicious code in the final page

Thanks for your attention!