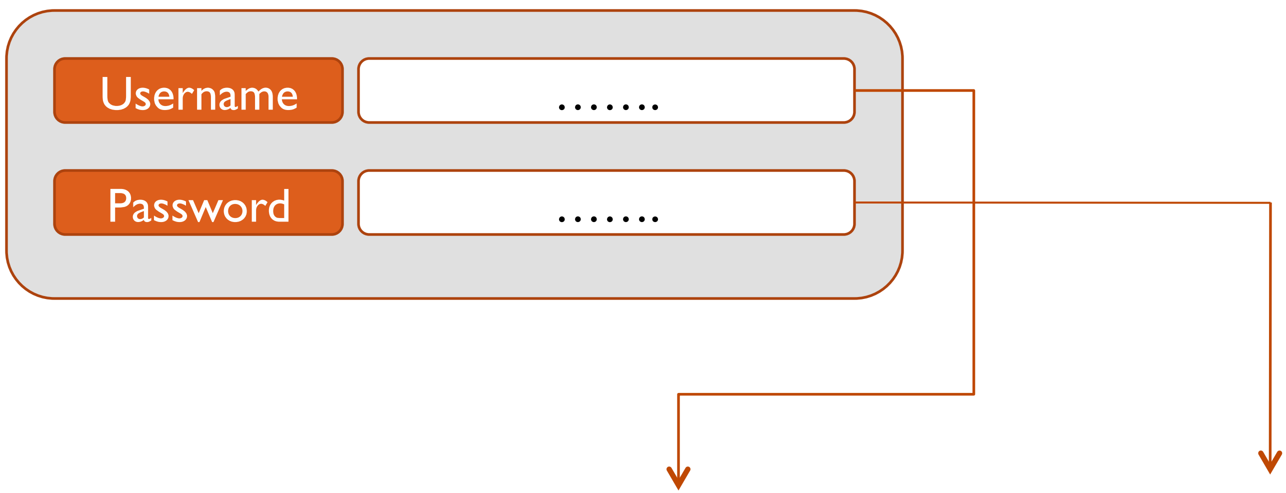
S3 : A Symbolic String Solver for Vulnerability Detection in Web Applications

Minh-Thai Trinh, Duc-Hiep Chu, Joxan Jaffar

National University of Singapore (NUS)

Web applications

- *Usually*:
  - take **string values** as inputs,
  - manipulate **string values**, and then
  - use **string values** to construct database queries.



- `"SELECT ... where user='$user' and password='$pwd'"`

Vulnerabilities in web applications

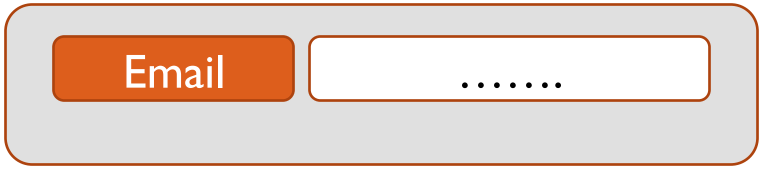
- From OWASP, the most serious web security vulnerabilities:
  - #1: Injection flaws such as SQL injection
  - #3: Cross Site Scripting (XSS) flaws

Due to inadequate sanitization and inappropriate use of input **strings** provided by users

Dynamic Symbolic Execution (DSE)

- Current trend to detect vulnerabilities in web applications (Saxena[SP'10], Brumley[SP'10,ICSE'14])
- How does it work?
  - Symbolic execution for high coverage of program execution space
  - But concretize when necessary to avoid false positive
    - Event space
    - Loops
    - Hard-to-solve constraints such as non-linear constraints

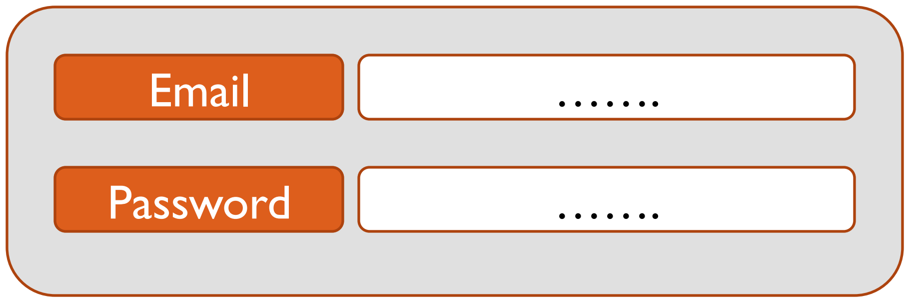
Email validation



Client-side JavaScript code

```
function validateEmail(email) {
  ...
  // break email into 3 parts
  // local part
  // @ character
  // domain part
  if (domain.equals("nus.edu.sg")){
    var reg = new RegExp("^[a-zA-Z][0-9]*$");
    var test1 = reg.test(local);
    var test2 = local.length == 8;
    return test1 && test2;
  }
  else if (domain.equals("comp.nus.edu.sg"))
    return local.length >= 4;
  else
    return false;
}
```

Server-side PHP code




```
$eml = $_POST['email'];
$pwd = $_POST['password'];
$stm="SELECT ... where email='$eml' and password='$pwd'";
$result = mysql_query($stm);
```

SQL injection?

- To detect SQL injection, we may want to test whether $eml contains the string:

```
' OR 1=1--
```

- The attack specification (e.g. above) is given by security experts

Dynamic Symbolic Execution (DSE)

- First express all the input email addresses that can be validated by using the symbolic constraints
  - So that we know the form of $eml at the server side
- Combine with the attack specification (on $eml) to decide if the JavaScript code is vulnerable to SQL injection

```
function validateEmail(email) {
  ...
  // break email into 3 parts
  // local part
  // @ character
  // domain part
  if (domain.equals("nus.edu.sg")){
    var reg = new RegExp("^[a-zA-Z][0-9]*$");
    var test1 = reg.test(local);
    var test2 = local.length == 8;
    return test1 && test2;
  }
  else if (domain.equals("comp.nus.edu.sg"))
    return local.length >= 4;
  else
    return false;
}
```

PC1

email = local .“@” . domain /\
domain = “nus.edu.sg” /\
reg = /^[a-zA-Z][0-9]*$/ /\
local in reg /\ len(local) = 8

PC2

email = local .“@” . domain /\
domain = “comp.nus.edu.sg” /\
len(local) >= 4

Vulnerability Detection ~> Constraint Solving

Email address that passes the validation **if** PC1 or PC2 is satisfiable

Email address that leads to SQL injection **if** It passes the validation and leads to $eml which contains the string `' OR 1=1--`

Checking satisfiability of formulae

- From vulnerability detection to checking the satisfiability of the following formulae:

PC1′

email = local ."@" . domain /\
domain = "nus.edu.sg" /\
reg = /^[a-zA-Z][0-9]*$/ /\
local in reg /\ len(local) = 8 /\
email = $eml /\
$eml contains "' OR 1=1--"

→ UNSAT

PC2′

email = local ."@" . domain /\
domain = "comp.nus.edu.sg" /\
len(local) >= 4 /\ email=$eml /\
email contains "' OR 1=1--"

→ email = ' OR 1=1@comp.nus.edu.sg

Traditional Random Testing

- Test with concrete inputs
  - To exploit the SQL injection vulnerability, the input email addresses need to be validated first
    - In order to reach $eml at the server side
  - E.g. `' OR 1=1--` → **Does not pass the validation test**
  - **Unlikely** to test with the interesting case:

```
' OR 1=1--@comp.nus.edu.sg
```

S3: A Robust and Efficient String Solver

S3 Language

- Independent of input languages, e.g. PHP, JavaScript, etc.
- Non-string constraints
  - E.g., constraints of int-sort, bool-sort, …
  - Length constraints
- String constraints over multiple string variables:
  - String equations
  - Membership predicates
  - String operations
    - ReplaceAll
- Regular expressions:
  - Constructed from Constant Strings using Union, Concatenation, Kleene star operations
  - S3 also supports character classes, escaped sequences, repetition operators, sub-match extraction using capturing parentheses, etc.

Comparison with Kaluza

- Kaluza is the representative for the state-of-the-art
  - Supports the most expressive constraint language so far
  - Is the underlying solver for a DSE framework (Kudzu[SP'10]) to detect vulnerabilities in JavaScript programs
  - Can also be used in other vulnerability analyses (NoTamper[CCS'10], WAPTEC[CCS'11])
- S3 is even more expressive:
  - Unbounded strings
  - High-level string operations such as ReplaceAll
    - Used frequently in sanitization
- S3 has better performance, better robustness

JavaScript Example

```
function validateFields(p1,p2) {
  var re1 = /^(ab)*$/;
  var re2 = /^(bc)*$/;
  var t1 = re1.test(p1);
  var t2 = re2.test(p2);
  var t3 = p2.length > 0;
  return (t1 && t2 && t3)
}
```

p1 . p2 ≠ **"abababababababcc"**

Constraint Solving

JavaScript Code

```
function validateFields(p1,p2)
{
  var re1 = /^(ab)*$/;
  var re2 = /^(bc)*$/;
  var t1 = re1.test(p1);
  var t2 = re2.test(p2);
  var t3 = p2.length > 0;
  return (t1 && t2 && t3)
}
```

p1 . p2 ≠ **"abababababbabcc"**

Generated Constraints

$$\texttt{p1} \in (\text{"}ab\text{"})^{*} \wedge$$
$$\texttt{p2} \in (\text{"}bc\text{"})^{*} \wedge$$
$$\textbf{length}(\texttt{p2}) > 0 \wedge$$

$$\texttt{p1} \cdot \texttt{p2} = \text{"}abababababbabcc\text{"}$$

Check for satisfiability. If UNSAT then the program is SAFE

Star representation

Generated Constraints		Our Internal Representation
$p1 \in (\text{"}ab\text{"})^* \wedge$	⇨	$p1 = \mathbf{star}(\text{"}ab\text{"}, n1) \wedge$
$p2 \in (\text{"}bc\text{"})^* \wedge$	⇨	$p2 = \mathbf{star}(\text{"}bc\text{"}, n2) \wedge$
$\mathbf{length}(p2) > 0 \wedge$		$\mathbf{length}(p2) > 0 \wedge$
$p1 \cdot p2 = \text{"}abababababcc\text{"}$		$p1 \cdot p2 = \text{"}abababababcc\text{"}$

Regular Expression to String Equation

- $\mathtt{p1} \in (\text{“}ab\text{”})^{\star} \rightarrow \mathtt{p1} = \mathbf{star}(\text{“}ab\text{”}, \mathtt{n1})$
- n1 is used to represent the number of repeating “ab”
- n1 is a variable, not a constant
- n1 is a fresh variable and generated automatically
- Specifically, **star**(“ab”, n1) can be interpreted as:
  - (p1 = “” /\ n1=0) V p1 = “ab” . star(“ab”, n1-1)
  - (p1 = “” /\ n1=0) V p1 = star(“ab”, n1-1) . “ab”
  - (p1 = “” /\ n1=0) V p1 = “ab” . star(“ab”, n1-2) . “ab”
- Guided by the current context

Incremental Solving of S3

Kaluza (Generate and Test)	S3
• Generate all possible length assignments for p1 and p2: • {pair | pair=(**len**(p1),**len**(p2))} = {(0, 12), (2, 10), (4, 8), (6, 6), (8, 4), (10, 2)} ↓ **6 times of testing**	• **star**("ab",n1) . **star**("bc",n2) = "ababababababcc" ↓ • **star**("ab",n1) . **star**("bc",n2-1) ."bc" = "ababababababcc" ↓ **UNSAT** (since "bc" ≠ "cc")

Note that len(p2) > 0

In summary

- Kaluza: **generate and test** approach
  - Generates all possible length assignments
  - For each length assignment, test if any string assignment satisfies the given formula.
  - Suffers from the combinatorial explosion
- S3: **incremental solving** approach

Implementation

- Is built on top of Z3-str (FSE'13) to exploit Z3's infrastructure
  - Lemma generation
  - Non-string constraints
- S3 is more expressive than Z3-str:
  - Regular expressions (e.g. /a*b*/)
  - Membership predicates (e.g. x is in /a*b*/)
  - String operations that work on regular expression (e.g. replaceAll, match, split, etc.)

Experimental Evaluation

- Kaluza benchmarks: 50000+ test cases
  - Generated from the vulnerability analysis of Kudzu[SP'10]
- Classified by Kaluza into 2 categories
  - SAT Category: 21819 benchmarks
  - UNSAT Category: 33230 benchmarks

Interpreting the solver's conclusions

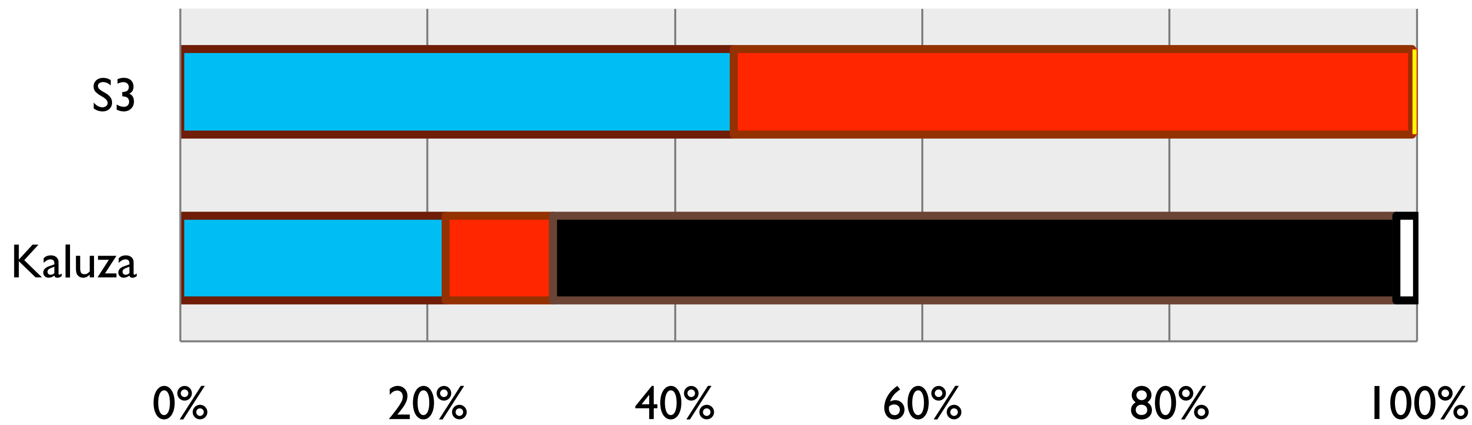
- SAT:
  - The formula is satisfiable
  - Can generate the test input to exploit the vulnerabilities
- UNSAT:
  - The formula is unsatisfiable
  - Cannot generate any test input to exploit the vulnerabilities
- MAYBE:
  - Inconclusive
  - Need further investigation

Experimental Evaluation

	Kaluza	S3
SAT	21808	21813
UNSAT	0	6
ERROR	11	0

S3 vs. Kaluza on SAT Category (21819 benchmarks)

Experimental Evaluation



	Kaluza	S3
UNSAT	7124	14877
SAT	2894	18210
ERROR	22653	
TO(1min)	559	
MAYBE		143

S3 vs. Kaluza on UNSAT Category (33230 benchmarks)

Experimental Evaluation

	#Files	Time(s)		
		K	S3	K/S3
SAT/Small	19984	5190	267	19.4x
SAT/Big	1835	3165	166	19.0x
UNSAT/Small	11761	4532	173	26.2x

Table 8: Timing Comparison: S3 vs. Kaluza

Conclusion

- A string solver
  - Support a **rich** set of constraints,
    - Generated from vulnerability analysis of web applications
  - **Robust** and **efficient**
- A *modular* contribution to any hypothetical DSE end-to-end system
- The tool is available soon

Future Work

- Strengthening the tool
  - Conflict clause learning in the string theory

- Integrating into an advanced DSE framework