A theory of injection-based vulnerabilities in formal grammars

Eric Alata, LAAS-CNRS
Pierre-François Gimenez, CentraleSupelec

GT MFS, March 28th, 2023

Opening example

Question time

Complete the following sentence:

Paris is to ␣ what London is to ␣.

Opening example

Question time

Complete the following sentence:

Paris is to ___ what London is to ___.

First kind of answer

- France and England
- Leads to: "Paris is to France what London is to England."
- Proposed by those who understand the intent behind the question

Opening example

Question time

Complete the following sentence:

Paris is to __ what London is to __.

First kind of answer

- France and England
- Leads to: "Paris is to France what London is to England."
- Proposed by those who understand the intent behind the question

Second kind of answer

- o crowded for you, and that's and me
- Leads to: "Paris is too crowded for you, and that's what London is to me."
- Proposed by those who know about injection attacks

What is an injection attack

Injection attack

An injection attack leverages a user input to modify the semantics of a sentence

What is an injection attack

Injection attack

An injection attack leverages a user input to modify the semantics of a sentence

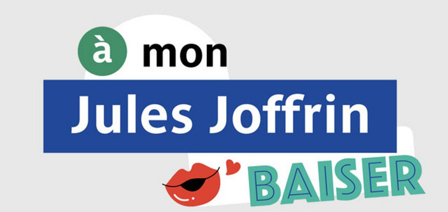
"The Voyage of Doctor Dolittle is canceled"

What is an injection attack

Injection attack

An injection attack leverages a user input to modify the semantics of a sentence

"The Voyage of Doctor Dolittle is canceled"



"À mon Jules Joffrin baiser"
"Jules Joffrin" is a Parisian subway station.
The whole sentence means "I give a kiss to my boyfriend"

And in software engineering?

SQL injection are well-known

A developer writes an authentication query:

```
SELECT id FROM user WHERE login='␣' AND password='␣'
```

If the user input is `admin` and `' OR 1=1--` it leads to:

```
SELECT id FROM user WHERE login='admin' AND password='' OR 1=1--'
```

Access granted, no need for the password!

And in software engineering?

SQL injection are well-known

A developer writes an authentication query:

```
SELECT id FROM user WHERE login='␣' AND password='␣'
```

If the user input is `admin` and `' OR 1=1--` it leads to:

```
SELECT id FROM user WHERE login='admin' AND password='' OR 1=1--'
```

Access granted, no need for the password!

Injection-based attacks concern not only SQL. . .

- Interpreted languages: bash, JavaScript, python
- Formats: JSON, XML
- Protocols: SMTP, LDAP
- Markup languages: HTML, CSS
- Even chatbots! (ChatGPT prompt injection)

What systems can be vulnerable?

Many systems process received instructions

- A browser receives and displays a page and executes scripts
- A database receives a query and applies it on its data
- A robot executes an order received though a network protocol

Injection vulnerabities

- These instructions may be structured using a query language, a protocol, etc.
- When instructions depend on user input, they are generally built by concatenation: it can lead to injection vulnerabilities
- Injections are a very serious threat:
    - #3 threat to web services according to OWASP
    - Appears 3 times in CWE Top 25 Most Dangerous Software Errors

What is this presentation about?

A formal approach

- Use the theory of formal language
- Propose a definition of injection vulnerabilities
- Propose two security properties and analyze their decidability
- Highlight some vulnerable language patterns

Formal language and grammar

The theory of formal languages studies the syntactic aspects of languages

Formal language

A formal language L is a set of valid strings called "words". Such string can be a SQL query, a C program, a network packet, etc.

Formal grammar

A grammar G describes a language $L(G)$ through a set of rewriting rules. If it is possible to rewrite the starting symbol into a word by applying rules, then this word is in the language described by that grammar.

Grammar and derivation

Starting symbol: <Query>

<Query> → **SELECT** <SelList> **FROM** <FromList> **WHERE** <Condition>

<SelList> → <Attribute> | <Attribute> **,** <SelList>

<FromList> → <Table> | <Table> **,** <FromList>

<Condition> → <Condition> **AND** <Condition> | <Attribute> **IN** **(** <Query> **)**

| <Attribute> **=** <Attribute>

Grammar and derivation

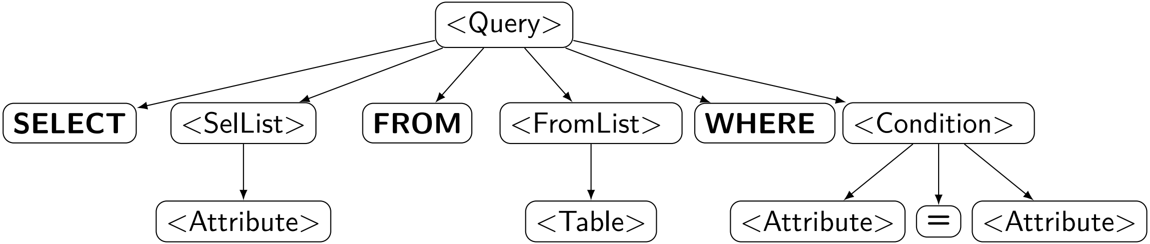
Starting symbol: <Query>

<Query> → **SELECT** <SelList> **FROM** <FromList> **WHERE** <Condition>

<SelList> → <Attribute> | <Attribute> **,** <SelList>

<FromList> → <Table> | <Table> **,** <FromList>

<Condition> → <Condition> **AND** <Condition> | <Attribute> **IN (** <Query> **)**
| <Attribute> **=** <Attribute>



Grammar and derivation

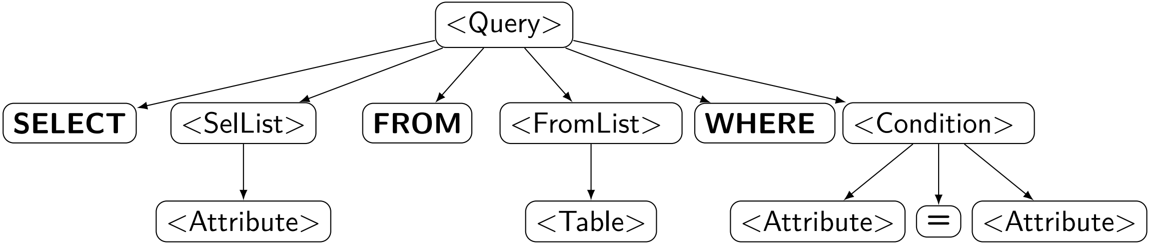
Starting symbol: <Query>

<Query> → **SELECT** <SelList> **FROM** <FromList> **WHERE** <Condition>

<SelList> → <Attribute> | <Attribute> **,** <SelList>

<FromList> → <Table> | <Table> **,** <FromList>

<Condition> → <Condition> **AND** <Condition> | <Attribute> **IN** **(** <Query> **)**
| <Attribute> **=** <Attribute>



<Query> $\Rightarrow^*$ **SELECT** <Attribute> **FROM** <Table> **WHERE** <Attribute> **=** <Attribute>

Grammar and language classes

Language and grammar classes

- Languages are grouped into classes depending on their properties. Simpler languages are easier to parse but have less expressive power.
- For each language class, there is generally a grammar class that generates it.

Informal presentation of some classical classes

- Regular language: languages that can be expressed with regular expression or finite-state automata
- Deterministic context-free language $\approx$ languages that can be parsed in linear time
- Context-free language: languages recognized by pushdown automata

Regular $\subset$ Deterministic $\subset$ Context-free

CentraleSupélec

1. Introduction
2. Background on formal language theory
3. **Formalization and security properties**
4. Conclusion and perspectives

Definitions

Query

A query is a complete command. For example: SQL query, JSON file, a network message, etc.

Template

- A fill-in-the-blanks template **t** is the set of strings written by the developer
- Example: `"SELECT ␣ FROM DB WHERE PRICE>␣ AND ID=22"`

Injection

- An injection is the set of strings that are inserted in a template
- Example: `"NUMBER"` and `"23.99"`
- Injections (always in red) may be legitimate or malicious

How to modelize a malicious injection?

Intent

- We assume that the developer has an intent in mind when they write the template.
- We modelize the intent with *a symbol or a sequence of symbol* denoted ι (for example: <Condition> or <Comparator> <Number>)
- **An injection w is legitimate if $\iota \Rightarrow^* w$**
- Languages and grammars don't deal with semantics... but compilers/interpreters do and rely on parsers, and parsers are based on grammars.
- It depends on the grammar and not only on the language!

Example

- Template: **SELECT** <Attribute> **FROM** <Table> **WHERE** <Attribute> **=** ␣
- Intent: <Attribute>
- Malicious injection: <Attribute> **AND** <Attribute>**=**<Attribute>

Intent-equivalence

Question

In which condition does a template $p \text{\textvisiblespace} s$ only accept legitimate injections?

Definitions

- First, we define the set of possible injections in this template :
  $F(L,(p,s)) = \{w \mid pws \text{ is a word of } L\}$
- Then, we define the set of injections that are expected by the developer :
  $E(G,\iota) = \{w \mid \iota \Rightarrow^* w\}$

Intent-equivalence

A template $p \text{\textvisiblespace} s$ is said to be *intent-equivalent* to ι if

$$S \Rightarrow^* p\iota s \qquad \text{and} \qquad F(L(G),(p,s)) = E(G,\iota)$$

i.e., if the intent can appear in $p \text{\textvisiblespace} s$ and the possible injections are all expected

Intent-equivalence results

- Intent-equivalence is decidable for regular and some deterministic grammars
- It is decidable for context-free grammars for terminal (non-derivable) intents, but undecidable with any intent.

	≥ 1 blanks $\boldsymbol{\iota} \in (\Delta)^m$	≥ 1 blanks $\boldsymbol{\iota} \in (\Delta^+)^m$	≥ 1 blanks $\boldsymbol{\iota} \in (T^+)^m$
Regular Visibly pushdown LR(0)	Decidable	Decidable	Decidable
LR(k)	Decidable	?	Decidable
Linear Context-free	Undecidable	Undecidable	Decidable

Is a template intent-equivalent to ι?

$\Rightarrow$ most programming languages can be checked for injection vulnerability by static analysis

Intent-security

Question

In which condition a grammar can only generate intent-equivalent templates?

Definitions

- Let us define the set of injection of a whole grammar for a particular intent : $I(G,\iota) = \bigcup_{\{(p,s) \mid S \Rightarrow^* p\iota s\}} F(L(G),(p,s))$
- The set of *unexpected injections* is the set of injections that may appear in a template and that is not explained by the intent : $\delta I(G,\iota) = I(G,\iota) - E(G,\iota)$

Intent-security

A grammar is intent-secure for the intent ι if $\delta I(G,\iota) = \varnothing$.

Example

There is a grammar G such that $L(G) = \{a^n cdb^n \mid n \geq 0\}$ that is intent-secure for all symbols

Intent-security

- No infinite regular language (and languages that include infinite regular sublanguages) have an intent-secure grammar
- For two blanks, no context-free language have an intent-secure grammar
- It is undecidable for one blank for deterministic grammars

	One blank	≥ 2 blanks
Finite, $\lvert L \rvert \geq 2$	Decidable	Decidable
Grammars with infinite regular sublanguage	False	False
Infinite LR(0), linear or context-free	Undecidable	False

Is a grammar intent-secure?

$\Rightarrow$ verifying whether a grammar is intent-secure is difficult, and most are in fact vulnerable!

Focus on infinite regular languages

No infinite regular language (and languages that include infinite regular sublanguages) have an intent-secure grammar

Idea behind the impossibility

- The formal proof is based on the pumping lemma, but can be explained in a different way.
- The only way to have an infinite regular expression is to have a repetition with *. For example, in SQL: **SELECT** (<Attribute> **,**)* <Attribute> **FROM** <Table> is an infinite regular expression.
- In the template **SELECT** ␣ **FROM** <Table>, one can inject **<Attribute>, <Attribute>** even if the intent is <Attribute>

Implication

It explains why so many languages are vulnerable: infinite regular patterns are ubiquitous! Another example: (**Condition OR**)* **Condition** (used in the SQL injection attacks)

Focus on infinite context-free languages

For two blanks, no context-free language has an intent-secure grammar

Example

- Template: **SELECT** <Attribute> **FROM** <Table> **WHERE** ␣ **IN (SELECT** <Attribute> **FROM** <Table> **) AND** <Attribute> **=** ␣
- Intents: two <Attribute>
- Malicious injection:
  - <Attribute> **IN (SELECT** <Attribute> **FROM** <Table> **WHERE** <Attribute>
  - <Attribute> **)**
- Completed sentence: **SELECT** <Attribute> **FROM** <Table> **WHERE** <Attribute> **IN (SELECT** <Attribute> **FROM** <Table> **WHERE** <Attribute> **IN (SELECT** <Attribute> **FROM** <Table> **) AND** <Attribute> **=** <Attribute> **)**

Focus on infinite context-free languages (cont.)

SELECT <Attribute> **FROM** <Table> **WHERE** <Attribute> **IN (SELECT** <Attribute> **FROM** <Table> **WHERE** <Attribute> **IN (SELECT** <Attribute> **FROM** <Table> **)** **AND** <Attribute> **=** <Attribute> **)**

Intuitively: with a recursive structure, one can add a level to the derivation tree by modifying both sides of the recursive structure

Implication

- This pattern is ubiquitous as well: any kind of recursive structure with tags, parenthesis, etc.
- This vulnerability needs blanks on both sides of the recursive structure
- Rarely seen in practice, but can happen in LDAP injection attacks

And more complex grammars?

Context-sensitive grammar

- Our definition of unexpected injections is designed for context-free grammar, but let's think about context-sensitive grammar...
- Let L be any context-free language, and $k \geq 1$. Then:

  $$L'_k = \{w(\#\#w)^k \mid w \in L\}$$

  is a context-sensitive grammar that is intent-secure for up to k blanks for $\iota \in T$
- Not practical, just a proof of concept...

$\Rightarrow$ more complex grammar classes can bring more security properties

Conclusion and perspectives

Conclusion

- It is generally possible to use static analysis to verify the absence of injection vulnerability in a template
- Grammar security is generally undecidable and most grammars are vulnerable
- Regular patterns with * should be avoided if they may contain a user input
- One should be vigilant with recursive structure if blanks can appear on both sides
- Generally, the more complex the grammar class, the more guarantee we can get

Perspectives

- Static analysis of filtering
- Black-box injection fuzzer
- Design principles for languages that are intent-secure for one blank