

# A formal study of injection vulnerabilities and some tools it will enable

---

Pierre-François Gimenez, CentraleSupélec  
Joint work with Eric Alata, LAAS-CNRS

SoSySec, February 19, 2021



# 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"

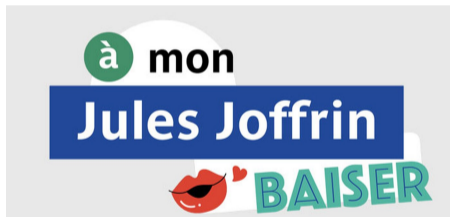
# What is an injection attack

## 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!



## 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!

### Injection-based attacks concern not only SQL...

- OS commands: Windows command line, bash
- Interpreted languages: JavaScript, python
- Formats: JSON, XML
- Protocols: SMTP, LDAP
- Markup languages: HTML, CSS

# 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 through a network protocol

## Injection vulnerabilities

- 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 very serious threats :
  - #1 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
- Formally define what is an injection attack
- Propose two security properties and analyze their decidability
- Highlight some vulnerable language patterns

## A few tools derived from the formal approach

- A secure query language, a fuzzer and an intrusion detection system
- The objective of a provisional ANR project proposal



- ① Introduction
- ② Background on formal language theory
- ③ Formalization and security properties
- ④ Results and implications
- ⑤ Ongoing and future projects

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 one can rewrite a starting symbol into a word by applying rules, then this word is in the language described by that grammar.

Each formal grammar describes one language, but each language can be described by several grammars:  $L(G) = L(G') \not\Rightarrow G = G'$

# Grammar and derivation

Starting symbol:  $\langle \text{Query} \rangle$

$\langle \text{Query} \rangle \rightarrow \mathbf{SELECT} \langle \text{SelList} \rangle \mathbf{FROM} \langle \text{FromList} \rangle \mathbf{WHERE} \langle \text{Condition} \rangle$

$\langle \text{SelList} \rangle \rightarrow \langle \text{Attribute} \rangle \mid \langle \text{Attribute} \rangle , \langle \text{SelList} \rangle$

$\langle \text{FromList} \rangle \rightarrow \langle \text{Table} \rangle \mid \langle \text{Table} \rangle , \langle \text{FromList} \rangle$

$\langle \text{Condition} \rangle \rightarrow \langle \text{Condition} \rangle \mathbf{AND} \langle \text{Condition} \rangle \mid \langle \text{Attribute} \rangle \mathbf{IN} ( \langle \text{Query} \rangle )$   
 $\mid \langle \text{Attribute} \rangle = \langle \text{Attribute} \rangle$

# Grammar and derivation

Starting symbol:  $\langle \text{Query} \rangle$

$\langle \text{Query} \rangle \rightarrow \mathbf{SELECT} \langle \text{SelList} \rangle \mathbf{FROM} \langle \text{FromList} \rangle \mathbf{WHERE} \langle \text{Condition} \rangle$

$\langle \text{SelList} \rangle \rightarrow \langle \text{Attribute} \rangle \mid \langle \text{Attribute} \rangle , \langle \text{SelList} \rangle$

$\langle \text{FromList} \rangle \rightarrow \langle \text{Table} \rangle \mid \langle \text{Table} \rangle , \langle \text{FromList} \rangle$

$\langle \text{Condition} \rangle \rightarrow \langle \text{Condition} \rangle \mathbf{AND} \langle \text{Condition} \rangle \mid \langle \text{Attribute} \rangle \mathbf{IN} ( \langle \text{Query} \rangle )$   
 $\mid \langle \text{Attribute} \rangle = \langle \text{Attribute} \rangle$

## Derivation example

$\langle \text{Query} \rangle \Rightarrow \mathbf{SELECT} \langle \text{SelList} \rangle \mathbf{FROM} \langle \text{FromList} \rangle \mathbf{WHERE} \langle \text{Condition} \rangle$

$\Rightarrow \mathbf{SELECT} \langle \text{Attribute} \rangle \mathbf{FROM} \langle \text{FromList} \rangle \mathbf{WHERE} \langle \text{Condition} \rangle$

$\Rightarrow \mathbf{SELECT} \langle \text{Attribute} \rangle \mathbf{FROM} \langle \text{Table} \rangle \mathbf{WHERE} \langle \text{Condition} \rangle$

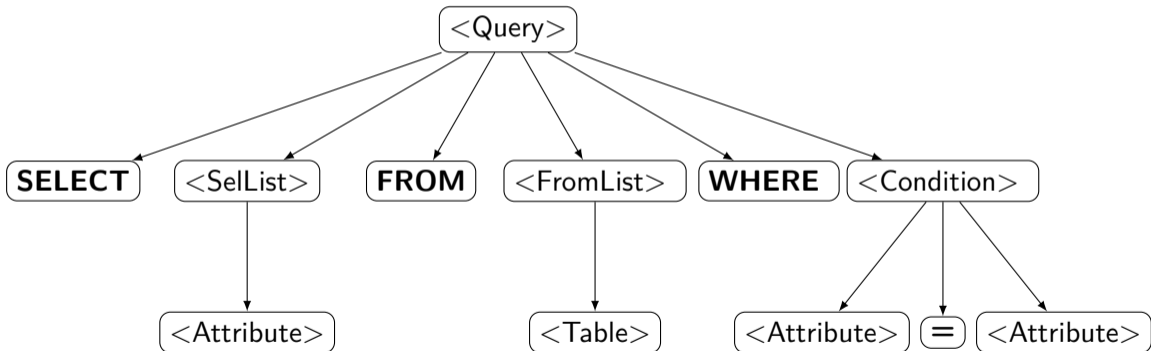
$\Rightarrow \mathbf{SELECT} \langle \text{Attribute} \rangle \mathbf{FROM} \langle \text{Table} \rangle \mathbf{WHERE} \langle \text{Attribute} \rangle = \langle \text{Attribute} \rangle$

We can also write directly:

$\langle \text{Query} \rangle \Rightarrow^* \mathbf{SELECT} \langle \text{Attribute} \rangle \mathbf{FROM} \langle \text{Table} \rangle \mathbf{WHERE} \langle \text{Attribute} \rangle =$   
 $\langle \text{Attribute} \rangle$

# Derivation tree

Derivation trees (= parse tree, concrete syntax tree) are another way of representing the set of rules used to derive a word





# Grammar and language classes

## Language classes

We can regroup languages into classes depending on their properties. Simpler languages are easier to parse but have less expressive power.

## Grammar classes

For each language class, there is generally a grammar class with some restrictions about the form of the rules so these grammars generate that language class.

## Informal presentation of some classical classes

- Regular language: all the languages that can be expressed with regular expression or finite-state automata
- Deterministic context-free language: languages that can be parsed in linear time
- Context-free language: languages whose words have parse trees



- 1 Introduction
- 2 Background on formal language theory
- 3 Formalization and security properties**
- 4 Results and implications
- 5 Ongoing and future projects

## Query

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

## Template

A fill-in-the-blanks template  $\mathbf{t}$  is the set of strings written by the developer. Example:

```
"SELECT ___ FROM DB WHERE PRICE>___ AND ID=22"
```

## Injection

A user input  $\mathbf{w}$  is the set of strings that is injected in a template. Example: "NUMBER" and "23.99". Injection may be legitimate or malicious. In **red**

For simplicity sake, examples in this talk will be restricted to template with a single blank

# How to modelize a malicious injection?

## Intent

- We assume that the developer has an intent in mind when they writes the template.
- We modelize the intent with *a symbol or a sequence of symbol* denoted  $\iota$  (for example:  $\langle \text{Condition} \rangle$  or  $\langle \text{Comparator} \rangle \langle \text{Number} \rangle$ )
- **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**  $\langle \text{Attribute} \rangle$  **FROM**  $\langle \text{Table} \rangle$  **WHERE**  $\langle \text{Attribute} \rangle = \_$
- Intent:  $\langle \text{Attribute} \rangle$
- Malicious injection:  $\langle \text{Attribute} \rangle$  **AND**  $\langle \text{Attribute} \rangle = \langle \text{Attribute} \rangle$

## Question

In which condition a template  $(p, s)$  can 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  $(p, s)$  is said to be *intent-equivalent* to  $\iota$  if

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

i.e. if the intent is possible in that place and if the possible injections are exactly the expected

## 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  $\iota$  if  $\delta I(G, \iota) = \emptyset$ .

## Example

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

# Inherently intent-(un)secure languages

The definitions of intent-equivalence and intent-security depend on a grammar

## Inherently intent-secure languages

- A language whose grammars are all intent-secure
- They don't exist: we can always craft an insecure grammar

## Inherently intent-insecure languages

- A language whose no grammar is intent-secure
- **SELECT \* FROM product WHERE price = \_\_\_**

Here, the intent can be a number. We can inject `123 OR availability="true"`. Since this injection works for all grammars, SQL is inherently intent-insecure.



- 1 Introduction
- 2 Background on formal language theory
- 3 Formalization and security properties
- 4 Results and implications**
- 5 Ongoing and future projects



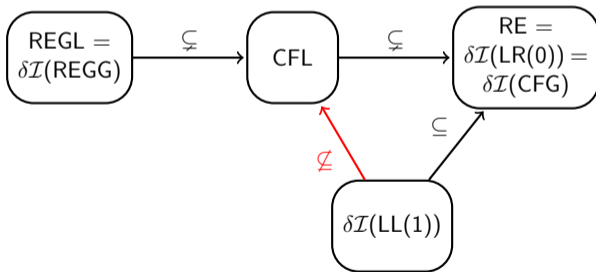
# Intent-equivalence

- 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.

	$\geq 1$ blanks $\iota \in (\Delta)^m$	$\geq 1$ blanks $\iota \in (\Delta^+)^m$	$\geq 1$ blanks $\iota \in (T^+)^m$
REGG	Decidable	Decidable	Decidable
VPG	Decidable	Decidable	Decidable
LR(0)	Decidable	Decidable	Decidable
LR( $k$ )	Decidable	?	Decidable
CFG	Undecidable	Undecidable	Decidable

# Injection characterization

- A language is regular iff it is the set of unexpected injections in a regular grammar
- Even simple grammars (LL(1)) can have complex (context-sensitive) injections
- A language can be described by a grammar iff it is the set of unexpected injections in a deterministic grammar



# Intent-security

- All infinite regular languages (and languages that include infinite regular sublanguages) are inherently intent-insecure
- For two blanks, all context-free languages are inherently intent-insecure
- It is undecidable for one blank for deterministic grammars

	One blank	$\geq 2$ blanks
Finite, $ L  \geq 2$	Decidable	Decidable
REGG with infinite language	False	False
Grammars with infinite regular sublanguage	False	False
LR(0) with infinite language	Undecidable	False
CFG with infinite language	Undecidable	False

# Intent-security

- All infinite regular languages (and languages that include infinite regular sublanguages) are inherently intent-insecure
- For two blanks, all context-free languages are inherently intent-insecure
- It is undecidable for one blank for deterministic grammars

	One blank	$\geq 2$ blanks
Finite, $ L  \geq 2$	Decidable	Decidable
REGG with infinite language	False	False
Grammars with infinite regular sublanguage	False	False
LR(0) with infinite language	Undecidable	False
CFG with infinite language	Undecidable	False

## Focus on infinite regular languages

*All infinite regular languages (and languages that include infinite regular sublanguages) are inherently intent-insecure*

### 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 inherently intent-insecure: infinite regular patterns are ubiquitous! Another example: **(Condition OR)\* Condition**

# Focus on infinite context-free languages

*For two blanks, all context-free languages are inherently intent-insecure*

## Idea behind the impossibility

- In infinite context-free languages, there exists  $A \Rightarrow^* w_1 b A c w_2$  ( $w_1$  and  $w_2$  may be empty)
- $A$  can be reached from the starting symbol  $S$ :  
 $S \Rightarrow^* p A s \Rightarrow^* p w_1 b A c w_2 s \Rightarrow^* p w_1 b w_1 b A c w_2 c w_2 s$ .
- Let the template be:  $p w_1 \_ A \_ w_2 s$ . The intents are  $b$  and  $c$ .
- We can inject  $b w_1 b$  and  $c w_2 c$  and get a valid word:  $p w_1 b w_1 b A c w_2 c w_2 s$ . It is an unexpected (malicious) injection
- Intuitively: with a recursive structure, one can add a level to the derivation tree by modifying both sides of the recursive structure

# Focus on infinite context-free languages (cont.)

## 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> )

## 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.

# Implications

- The problem does not stem from bad development practice but from the languages themselves
- Template analysis (intent-equivalence) is possible for deterministic grammars but not for more complex grammars, but may require asking or guessing the developer's intent

## Implications on language design

- It is possible (but not easy) to have intent-secure grammars
- Simpler languages are *not* more secure. On the contrary!
- 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
- More complex, context-sensitive languages could be safe with two blanks or more
- Finite language are probably the most secure





- 1 Introduction
- 2 Background on formal language theory
- 3 Formalization and security properties
- 4 Results and implications
- 5 Ongoing and future projects

# How to design an intent-secure language

## Problem

- Intent-security is either false or undecidable for most grammar classes
- Could we find sufficient conditions for a grammar to be intent-secure?
- What would such language look like?

## Sufficient conditions and necessary conditions

We found various sufficient conditions and necessary conditions. For example:

- An intent-secure grammar cannot have rules in the form  $A \rightarrow \alpha B \beta$  and  $A \rightarrow \alpha \delta \beta$  (because  $B$  could be replaced with  $\delta$ )
- A grammar that is LL(1), RR(1), whose each expected injections set is prefix-free and suffix-free, and with no rule  $A \rightarrow B$ , is intent-secure for one blank and an intent of length 1.

# How to design an intent-secure language

## Proof of concept with SQL

We developed a proof of concept named SeQreL (secure-L) that is intent-secure for one blank and an intent of length 1. Compare for example:

- SQL: `SELECT ___ AS Orders, Min(Price) FROM Customers`
- SeQreL: `SELECT < AS[Orders,___], [MIN[(Price)]]> FROM Customers`

## Ongoing work

- This language is context-free so it is not intent-secure with two blanks → we need to extend our definitions to context-sensitive languages
- In real-world applications, no need to have intent-security for all symbols
- We are looking for an actual case study: if you need to develop a DSL (domain-specific language) that would benefit of being intent-secure, contact me!

# A black-box injection fuzzer

## Fuzzers

- A fuzzer is a testing tool that sends user input to a system to find its vulnerabilities
- Only a few injection-focused fuzzers, like sqlmap

## Idea

- When the grammar and the template is known, it is easy to compute the set of injections
- In a black-box setting, the template could be inferred from the system answer

## Poirot

- A universal black-box injection fuzzer that analyzes which injections are syntactically correct to infer the template (with any grammar)
- Based on an A\* search, guided by an heuristic → theoretical guarantees
- Experiments with SQL, LDAP, XML, Bash and SMTP → still some performance issues

## Injection IDS

- Injection IDS exist but are rarely used because of their complexity
- They require source code or library modification, developer input, etc.
- We work on an IDS with minimal configuration and interaction: "plug and play"
- In particular: we don't assume access to the source code and we don't taint the template

## Idea

- The IDS is a proxy placed between the back-end server and the database
- The difficulty is to identify the injection inside the query
- In development environment, the templates and their intents are inferred from queries
- In production environment, the query are verified with the learned templates. We can raise an alert or cancel the query if needed



# Provisional ANR project proposal

Objective: offer a methodological and technical toolbox to prevent, identify and mitigate injection threat in vulnerable systems

## Main focuses

- Broaden the range of injection logic (insertion, overwrite, mixed)
- Take advantage of more elaborate theoretical constraints on interpreted languages (context-sensitive languages)
- Investigate possibility to take advantage of interpreter operational semantics to assess threats and mitigate them
- Investigate chained injection problems (controllability of the final interpreter)

## Current team

- Pierre-François Gimenez, CentraleSupélec
- Eric Alata, LAAS-CNRS
- Benoît Morgan, IRIT
- Thomas Robert, Télécom Paris

# Conclusion

- Injection vulnerabilities are not specific to SQL but are present in all kinds of languages and systems that handle user input within structured data
- Injection vulnerabilities do not stem from poor programming skills but from deep flaws in ubiquitous patterns, such as infinite regular expression and recursive bracket-based expression
- New tools are possible to detect, remove or limit these vulnerabilities
- **Contact us! We are looking for industrial partnerships and research collaborations.** [pierre-francois.gimenez@centralesupelec.fr](mailto:pierre-francois.gimenez@centralesupelec.fr)

And thank you for your attention!