Towards programming languages free of injection-based vulnerabilities by design

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

LangSec workshop, May 15th, 2025

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

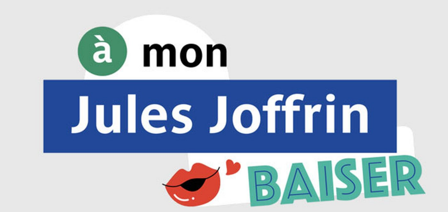
An injection attack leverages a user input to modify the semantics of a sentence

What is an injection attack

Injection attack

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

Paris subway station pun



à mon Jules Joffrin baiser

"Jules Joffrin" is a proper name
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 are not only about SQL...

- Interpreted languages: bash, JavaScript, python
- Formats: JSON, XML
- Protocols: SMTP, LDAP
- Markup languages: HTML, CSS

A very common and very serious threat in cybersecurity

What is this presentation about?

A formal approach based on languages

- Propose a definition of injection vulnerabilities
- Propose two security properties and analyze their decidability
- Highlight some vulnerable language patterns
- Propose design principles to create secure-by-design languages

Formalization and security properties

Running example

LDAP protocol

- LDAP is a widely used protocol for search in directory services
- It is regularly used for authentication

A simplified grammar (where `s` is any string):

$$S \to \texttt{(!}S\texttt{)} \qquad S \to \texttt{(s=s)} \qquad S \to \texttt{(\&}L\texttt{)}$$
$$S \to \texttt{(|}L\texttt{)} \qquad L \to S \qquad L \to LS$$

Examples

- `(&(uid=foo)(passwd=bar))`
- `(&(uid=foo)(passwd=bar)(!(status=online)))`
- `(|(mode=root)(&(uid=foo)(passwd=bar))`

Definitions

Query

A query is a complete command. For example: LDAP query, JSON file, a network packet, etc.

Template

- A fill-in-the-blanks template is the string written by the developer
- Example: `(&(uid=__)(passwd=1234))`

Injection

- An injection is the string inserted in a template
- Example: "`foo`"
- Injections (always in red) may be legitimate or malicious

How to modelize a malicious injection?

Intent

- We assume the developer has an *intent* in mind when they write the template
- The intent is modelized as *a symbol or a sequence of symbol* denoted ι (for example: `L` or `s = s`)
- **An injection w is legitimate if $\iota \Rightarrow^* w$**

Example

- Template: `(&(uid=␣)(passwd=1234))`
- Intent: `s`
- Legitimate injection: `root`, leading to `(&(uid=root)(passwd=1234))`
- Malicious injection: `foo)(loc=bar`, leading to `(&(uid=foo)(loc=bar)(passwd=1234))`

Intent-equivalence

Question

In which condition does a template p ␣ s only accept legitimate injections?

Definitions

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

A template p ␣ s is said to be *intent-equivalent* to ι if

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

Examples for LDAP

- `(!(uid=foo)`␣ is intent-equivalent to `)` → this template is secure
- `(&(uid=`␣`)(passwd=1234))` is *not* intent-equivalent to `s` → this template is not secure

Intent-equivalence results

- Decidable for regular and some deterministic grammars
- Decidable for context-free grammars for terminal intents, but undecidable with any intent

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

Is a template intent-equivalent to ι?

$\Rightarrow$ **most templates can be checked for injection vulnerability by static analysis**

Intent-security

Question

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

Definitions

- 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* (i.e., 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 $\delta I(G,\iota) = \varnothing$.

These definitions can be extended to multiples blanks as well.

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) or context-free	Undecidable	**False**

Is a grammar intent-secure?

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

Focus on infinite regular languages

No infinite regular language has an intent-secure grammar

Intuition behind the impossibility

- 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>
- It is closely related to the *pumping lemma*

Focus on infinite regular languages

Infinite regular patterns are ubiquitous!

- SQL: (<Condition> OR)* <Condition>
- SQL: (<Query> ;)* <Query>
- OS command: (<Command> ;)* <Command>
- OS parameters: (--<Parameter>)*
- SMTP: (<Email> %0A cc:)* <Email>
- JSON: (<Var> = <Value> ,)* <Var> = <Value>
- LDAP: (&((s = s))*)

$\rightarrow$ Many injection attacks rely on this vulnerability

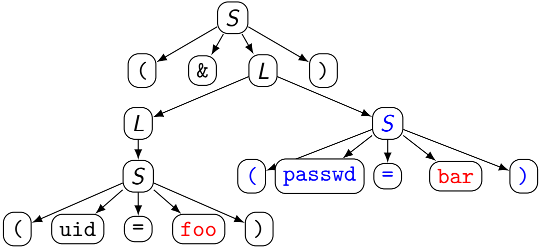
Focus on infinite context-free languages

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

Intuition behind the impossiblity

- Based on the pumping lemma: by modifying the query in two positions, one can shift down part of the parse tree
- Consider the LDAP query: `(&(uid=␣)(passwd=␣))`
- Legitimate injection: `(&(uid=foo)(passwd=bar))`
- Malicious injection: `(&(uid=admin)(!(&(1=0)(passwd=text))))`
- This LDAP attack is an actual injection used by attackers to bypass authentication

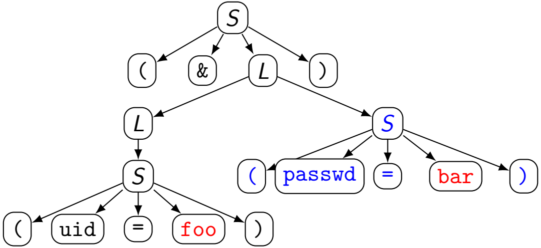
Example with an LDAP attack

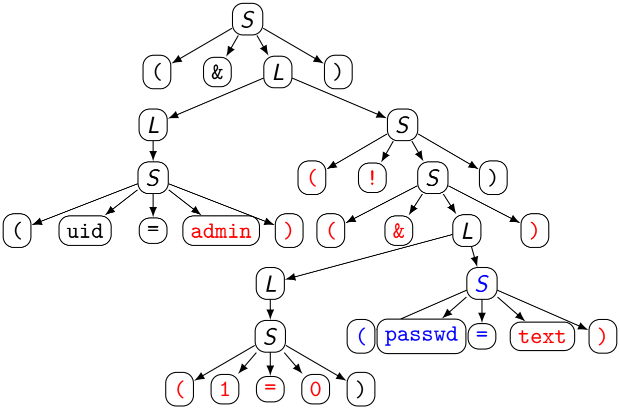


Legitimate injection

(&(uid=foo)(passwd=bar))

Example with an LDAP attack





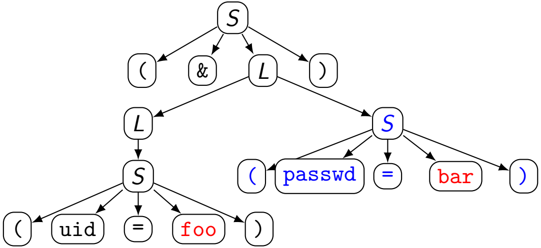
Legitimate injection

`(&(uid=foo)(passwd=bar))`

Malicious injection

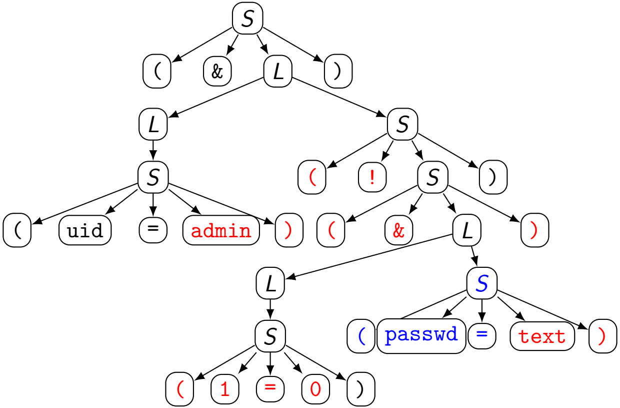
`(&(uid=admin)(!(&(1=0)(passwd=text))))`

Example with an LDAP attack



Legitimate injection

(&(uid=foo)(passwd=bar))



Malicious injection

(&(uid=admin)(!(&(1=0)(passwd=text))))

$\Rightarrow$ The blue subtree has been moved by the attack

Secure-by-design languages

Secure-by-design languages

Proving is difficult

- It is undecidable to prove that a deterministic grammar is intent-secure
- How to create languages that are secure by design?
- We only focus on intent-security for **one blank** (all deterministic languages are vulnerable with two blanks)

Base theorem

LLRR Theorem

Let $G = (T, N, R, S)$ a context-free formal grammar. Let denote G_A the grammar (T, N, R, A) where $A \in N$ and $L(G)$ the language described by a grammar G. Denote R_A the set of rules whose left-hand side is A. If

- G is $LL(1)$
- G is $RR(1)$
- G is epsilon-free, i.e. there are no rules of the form $A \to \epsilon$
- $L(G_A)$ is bifix-free (prefix-free and suffix-free) for all $A \in N$
- For all $A \in N$, if there exists $B \in \Delta$ and $\alpha \in \Delta^*$ such that $A \to B$ and $A \to \alpha$, then $\alpha = B$

Then G is intent-secure.

Secure design pattern

"Open-close" pattern

Informally: if every rule starts and ends with a unique terminal, then the grammar is intent-secure

Secure design pattern

"Open-close" pattern

Informally: if every rule starts and ends with a unique terminal, then the grammar is intent-secure

Let us consider a list written as $e_1, e_2, e_3, \dots e_n$. Its grammar is:

$$L \rightarrow e, L \quad L \rightarrow e$$

Secure design pattern

"Open-close" pattern

Informally: if every rule starts and ends with a unique terminal, then the grammar is intent-secure

Let us consider a list written as $e_1, e_2, e_3, \dots e_n$. Its grammar is:

$$L \rightarrow e, L \quad L \rightarrow e$$

This is a regular language, so it is vulnerable. Example:

$$e_1, \text{\textvisiblespace}, e_2$$

can be injected with e or e, e'

Example

We can modify the grammar by adding unique tags at the start and the end of each rule:

$$L \to [e, L] \qquad L \to <e>$$

Example

We can modify the grammar by adding unique tags at the start and the end of each rule:

$$L \rightarrow [e, L] \qquad L \rightarrow < e >$$

Try to inject something that is not just an element:

<␣>

Example

We can modify the grammar by adding unique tags at the start and the end of each rule:

$$L \to [e, L] \quad L \to <e>$$

Try to inject something that is not just an element:

```
<␣>
```

```
[1, [2, [␣, <4>]]]
```

Example

We can modify the grammar by adding unique tags at the start and the end of each rule:

$$L \to [e, L] \quad L \to <e>$$

Try to inject something that is not just an element:

```
<␣>
```

```
[1, [2, [␣, <4>]]]
```

The only possible injections are elements!

Example

We can modify the grammar by adding unique tags at the start and the end of each rule:

$$L \rightarrow [e, L] \qquad L \rightarrow <e>$$

Try to inject something that is not just an element:

```
<␣>
```

```
[1, [2, [␣, <4>]]]
```

The only possible injections are elements!

With a low cost, we can remove vulnerability against injections using one blank

Secure design pattern

"Open" pattern

Informally: if we assume that injection cannot target tags (because the user are typically not supposed to enter delimiters), then it is enough to have a unique opening tag.

Secure design pattern

"Open" pattern

Informally: if we assume that injection cannot target tags (because the user are typically not supposed to enter delimiters), then it is enough to have a unique opening tag.

We can apply a similar approach to secure LDAP (added tags in blue):

$$
\begin{array}{lll}
S \to (\,!\,S) & S \to \{\mathtt{s=s}\} & S \to (\&L) \\
S \to (\,|\,L) & L \to \,<S>\, & L \to [LS]
\end{array}
$$

Secure design pattern

"Open" pattern

Informally: if we assume that injection cannot target tags (because the user are typically not supposed to enter delimiters), then it is enough to have a unique opening tag.

We can apply a similar approach to secure LDAP (added tags in blue):

$$
\begin{array}{lll}
S \to \texttt{(!}S\texttt{)} & S \to \texttt{\{s=s\}} & S \to \texttt{(\&}L\texttt{)} \\
S \to \texttt{(|}L\texttt{)} & L \to \texttt{<}S\texttt{>} & L \to \texttt{[}LS\texttt{]}
\end{array}
$$

Earlier, we attacked the following template:

$$\texttt{(\&(uid=foo)(passwd=\textvisiblespace))}$$

Secure design pattern

"Open" pattern

Informally: if we assume that injection cannot target tags (because the user are typically not supposed to enter delimiters), then it is enough to have a unique opening tag.

We can apply a similar approach to secure LDAP (added tags in blue):

$$S \rightarrow (\,!\,S) \qquad S \rightarrow \{\texttt{s=s}\} \qquad S \rightarrow (\&L)$$
$$S \rightarrow (\,|\,L) \qquad L \rightarrow <S> \qquad L \rightarrow [LS]$$

Earlier, we attacked the following template:

```
(&(uid=foo)(passwd=␣))
```

It can be rewritten into:

```
(&[<{uid=foo}>{passwd=␣}])
```

Conclusion and perspectives

Conclusion and perspectives

Conclusion

- It is generally possible to statically verify the vulnerability of a template
- Regular patterns with * should be avoided if they may contain a user input
- All context-free grammars are vulnerable with two injections points
- Surprisingly, the more complex the grammar class, the more guarantees we can get
- We can create design principles for languages to make them intent-secure for one blank

Perspectives

If you are interested in applying these techniques to an actual language, contact me!