

Fixing Injection Vulnerabilities at the Root: Design Patterns for Secure Programming Languages

Eric Alata, LAAS-CNRS

Pierre-François Gimenez, Inria

Work supported by PEPR REV



THCon, May 6th, 2026





What is an injection attack

Injection attack

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

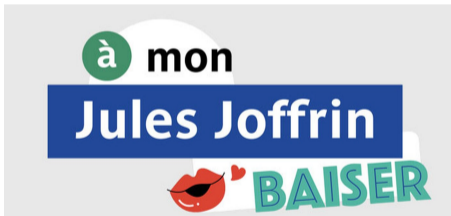


What is an injection attack

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



How to protect?

The classical ways to protect

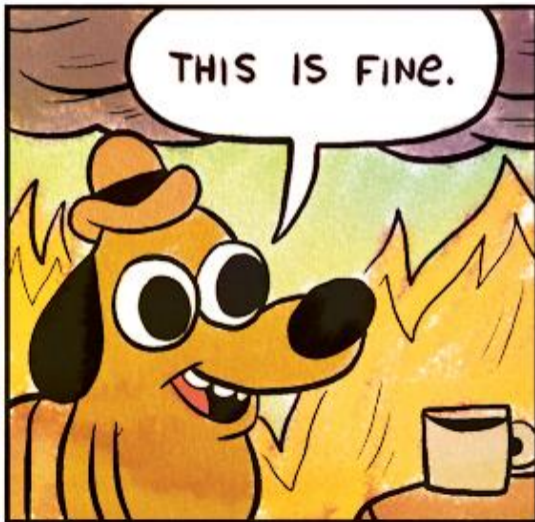
- Use a framework that handles it correctly
- Prepared statements for SQL

But...

- It is not possible for all languages (how to protect against Bash command injection?)
- It adds yet another overhead
- What if a dev (or an AI) does not use any of these protections?



"But the problem is solved!"





Let's extinguish the fire

The root issue

- The vulnerability does *not* come from careless developers or from unsecured frameworks
- The vulnerability is embedded in the languages *themselves*

Our approach

Classical scientific approach:

- Define the problem
- Analyze its properties
- Fix the problem *at the root*



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

Examples

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

A simplified grammar (where s is any string):

$$\begin{array}{lll} S \rightarrow (!S) & S \rightarrow (s=s) & S \rightarrow (&L) \\ S \rightarrow (|L) & L \rightarrow S & L \rightarrow LS \end{array}$$



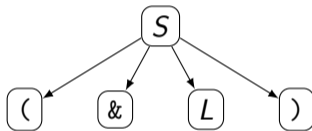
A derivation in detail

S



A derivation in detail

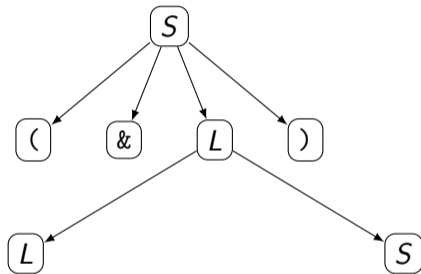
- $S \rightarrow (&L)$





A derivation in detail

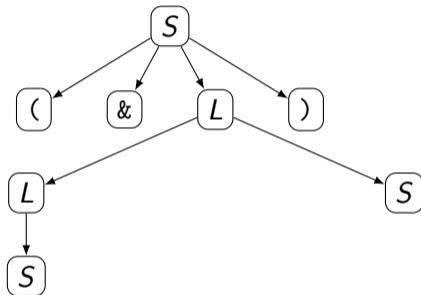
- $S \rightarrow (&L)$
- $L \rightarrow LS$





A derivation in detail

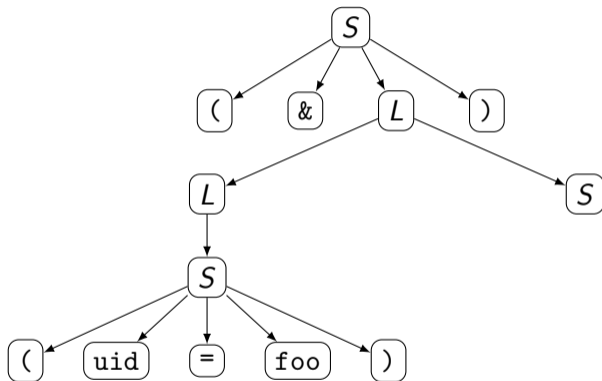
- $S \rightarrow (&L)$
- $L \rightarrow LS$
- $L \rightarrow S$





A derivation in detail

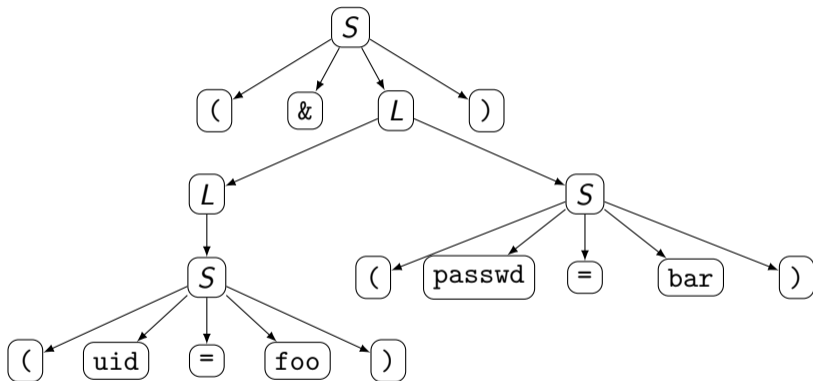
- $S \rightarrow (\&L)$
- $L \rightarrow LS$
- $L \rightarrow S$
- $S \rightarrow (s=s)$





A derivation in detail

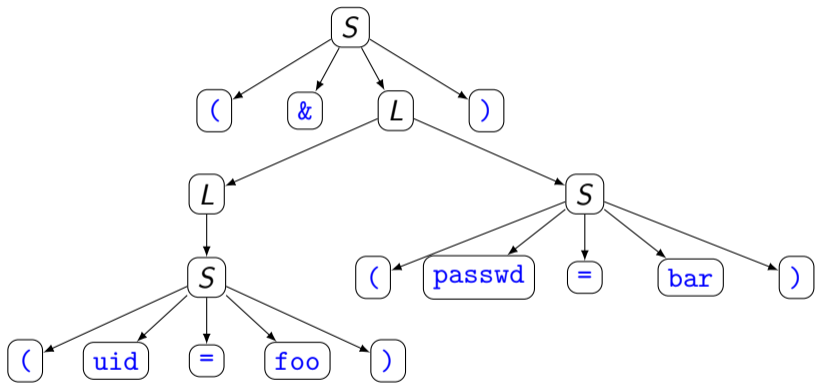
- $S \rightarrow (&L)$
- $L \rightarrow LS$
- $L \rightarrow S$
- $S \rightarrow (s=s)$
- $S \rightarrow (s=s)$





A derivation in detail

- $S \rightarrow (&L)$
- $L \rightarrow LS$
- $L \rightarrow S$
- $S \rightarrow (s=s)$
- $S \rightarrow (s=s)$



The **leaves** contain the query: `(&(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?

Definition

A template $p _ s$ is said to be *intent-equivalent* to ι if all text that can be included in the template can be derived from ι

Examples for LDAP

- $(!(uid=foo)_)$ is intent-equivalent to $_$ \rightarrow this template is secure
- $(\&(uid=_)(passwd=1234))$ is *not* intent-equivalent to s \rightarrow 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 ?

⇒ **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?

Definition

A grammar is *intent-secure* if it can only produce intent-equivalent templates.

Example with LDAP

- `(&(uid=___)(passwd=1234))` is *not* intent-equivalent to `s` → this template is not secure
- Therefore, LDAP is *not* intent-secure



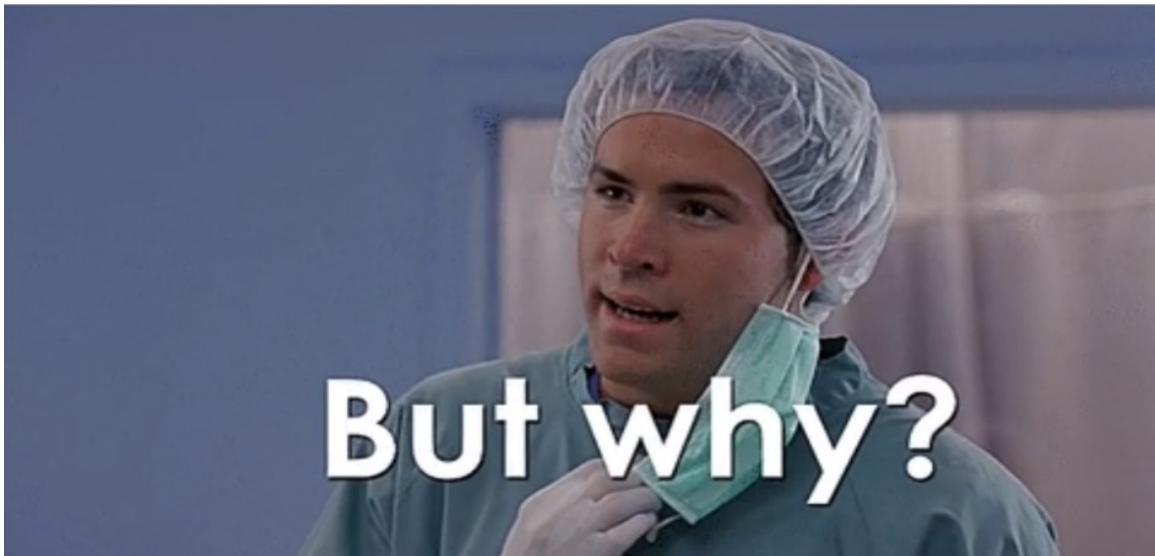
Intent-security

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

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

Is a grammar intent-secure?

⇒ **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: $(\langle \text{Condition} \rangle \text{ OR})^* \langle \text{Condition} \rangle$
- SQL: $(\langle \text{Query} \rangle ;)^* \langle \text{Query} \rangle$
- OS command: $(\langle \text{Command} \rangle ;)^* \langle \text{Command} \rangle$
- OS parameters: $(--\langle \text{Parameter} \rangle)^*$
- SMTP: $(\langle \text{Email} \rangle \%0A \text{ cc:})^* \langle \text{Email} \rangle$
- JSON: $(\langle \text{Var} \rangle = \langle \text{Value} \rangle ,)^* \langle \text{Var} \rangle = \langle \text{Value} \rangle$
- LDAP: $(\&((s = s))^*)$

→ Many injection attacks rely on this vulnerability



Focus on programming languages

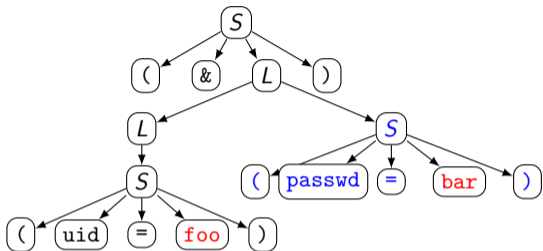
For two blanks, no programming language has an intent-secure grammar

Intuition behind the impossibility

- 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

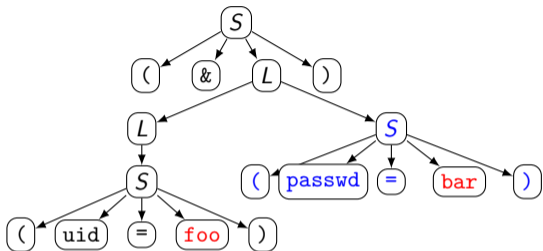


Legitimate injection

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

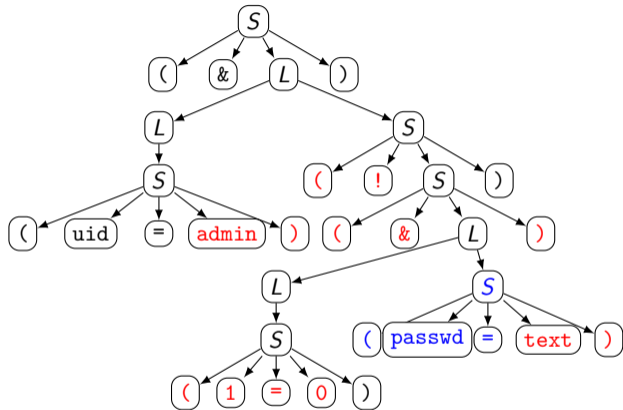


Example with an LDAP attack



Legitimate injection

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

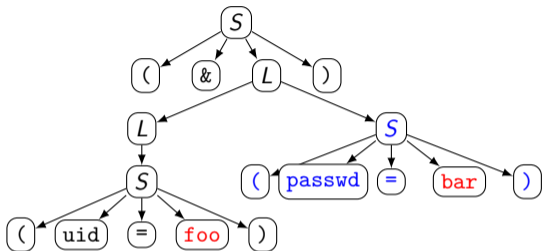


Malicious injection

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

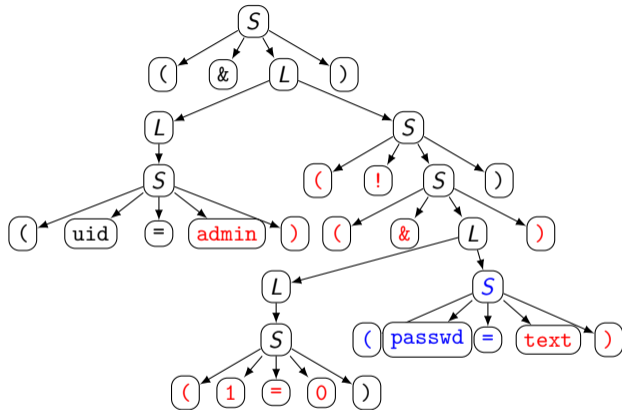


Example with an LDAP attack



Legitimate injection

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



Malicious injection

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

⇒ The blue subtree has been moved by the attack: this vulnerability **cannot be avoided**



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 could propose sufficient conditions for a language to be intent-secure
- 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 $A \rightarrow \epsilon$
- $L(G_A)$ 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 $A \rightarrow B$ and $A \rightarrow \alpha$, then $\alpha = B$

Then G is intent-secure.



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 $A \rightarrow \epsilon$
- $L(G_A)$ 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 $A \rightarrow B$ and $A \rightarrow \alpha$, then $\alpha = B$

Then G is intent-secure.

Ok, but in practice?



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:

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

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:

$$L \rightarrow [e, L] \quad L \rightarrow \langle e \rangle$$



Example

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

$$L \rightarrow [e, L] \quad L \rightarrow \langle e \rangle$$

Try to inject something that is not just an element:

$\langle _ \rangle$



Example

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

$$L \rightarrow [e, L] \quad L \rightarrow \langle e \rangle$$

Try to inject something that is not just an element:

$\langle _ \rangle$

$[1, [2, [_ , \langle 4 \rangle]]]$



Example

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

$$L \rightarrow [e, L] \quad L \rightarrow \langle e \rangle$$

Try to inject something that is not just an element:

$\langle _ \rangle$

$[1, [2, [_ , \langle 4 \rangle]]]$

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:

$$L \rightarrow [e, L] \quad L \rightarrow \langle e \rangle$$

Try to inject something that is not just an element:

$\langle _ \rangle$

$[1, [2, [_, \langle 4 \rangle]]]$

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):

<i>Before:</i>			<i>After:</i>		
$S \rightarrow (!S)$	$S \rightarrow (s=s)$	$S \rightarrow (&L)$	$S \rightarrow (!S)$	$S \rightarrow \{s=s\}$	$S \rightarrow (&L)$
$S \rightarrow (L)$	$L \rightarrow S$	$L \rightarrow LS$	$S \rightarrow (L)$	$L \rightarrow \langle S \rangle$	$L \rightarrow [LS]$



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):

<i>Before:</i>			<i>After:</i>		
$S \rightarrow (!S)$	$S \rightarrow (s=s)$	$S \rightarrow (&L)$	$S \rightarrow (!S)$	$S \rightarrow \{s=s\}$	$S \rightarrow (&L)$
$S \rightarrow (L)$	$L \rightarrow S$	$L \rightarrow LS$	$S \rightarrow (L)$	$L \rightarrow \langle S \rangle$	$L \rightarrow [LS]$

Earlier, we attacked the following template:

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



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):

<i>Before:</i>			<i>After:</i>		
$S \rightarrow (!S)$	$S \rightarrow (s=s)$	$S \rightarrow (&L)$	$S \rightarrow (!S)$	$S \rightarrow \{s=s\}$	$S \rightarrow (&L)$
$S \rightarrow (L)$	$L \rightarrow S$	$L \rightarrow LS$	$S \rightarrow (L)$	$L \rightarrow \langle S \rangle$	$L \rightarrow [LS]$

Earlier, we attacked the following template:

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

It can be rewritten into:

```
(&[<{uid=foo}>{passwd=___}])
```



Conclusion



Conclusion

Conclusion

- Injection vulnerabilities stem from the language itself!
- Regular patterns with * should be avoided if they may contain a user input
- All programming languages are vulnerable with two injection points
- Simple is not always more secure: 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
- It works for LDAP but not only!