Using tokens for secrets search or imitating SAST

Nikolai **KHECHUMOV**
Senior Security Engineer, Avito

Moscow, August 25, 2022

Meta (not that one) and Context

Nikolai **KHECHUMOV**
Senior Security Engineer @ Product Security Team

ntoskrnl
khechumov

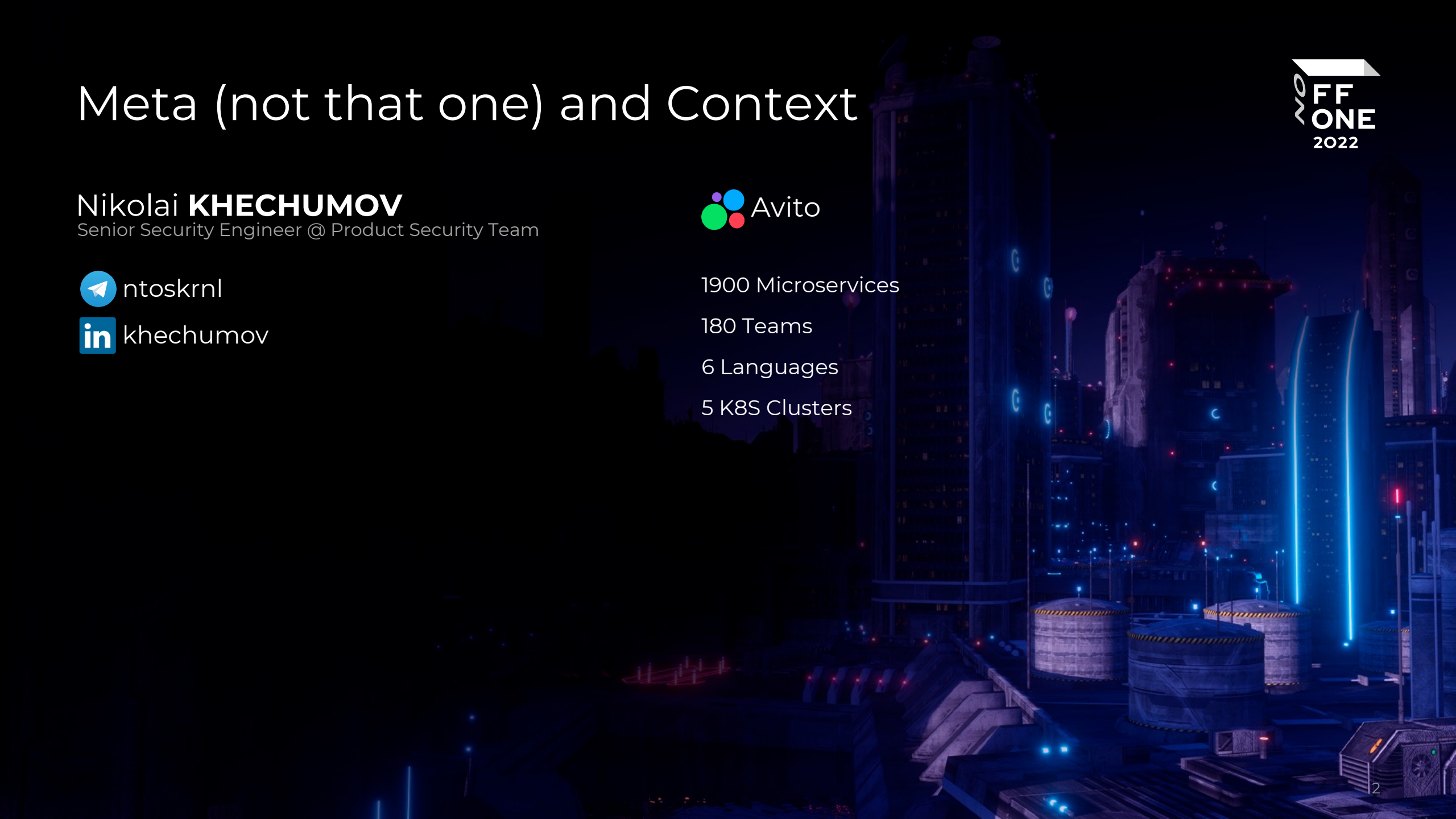
Avito

1900 Microservices

180 Teams

6 Languages

5 K8S Clusters



Agenda

- Few words about code security pipeline at Avito
- Problems of finding secrets in code
- Reasonable ways to solve
- Entire approach and results

Avito’s
Code Security Pipeline

Avito’s Code Security Pipeline

General Architecture

- Works in parallel with CI
- Triggered by VCS via hooks (pre-receive, post-receive, pr-open)
- Scans every push with a bunch of scanners, eq:
  - Language-dependent scanner (CodeQL, semgrep, RIPS, etc.)
  - Vulnerable / Malware Dependencies scanner
  - Secrets scanner (we are here)

- Extencible, no vendor-lock
- Tracks every finding from any scanner across git states and branches
  - Single finding format and shared deduplication strategies
  - State-machine detects specific lifecycle **events** and runs **reactions**
- Integrated with in-house SOAR (without defectdojo ☺)

- Topic for a separate talk

Eq. 0

- REACHED ORIGIN
- FOUND IN DIFF
- MARKED FALSE BY DEV
- APPR. FALSE POSITIVE

Eq. 1

- REACHED ORIGIN
- FOUND IN DIFF
- REACHED DEFAULT BRANCH
- RELEASED TO STAGING
- FIXED IN BRANCH A
- FIXED IN DEFAULT BRANCH
- DISAPPEARED

Avito’s Code Security Pipeline

Secrets Management

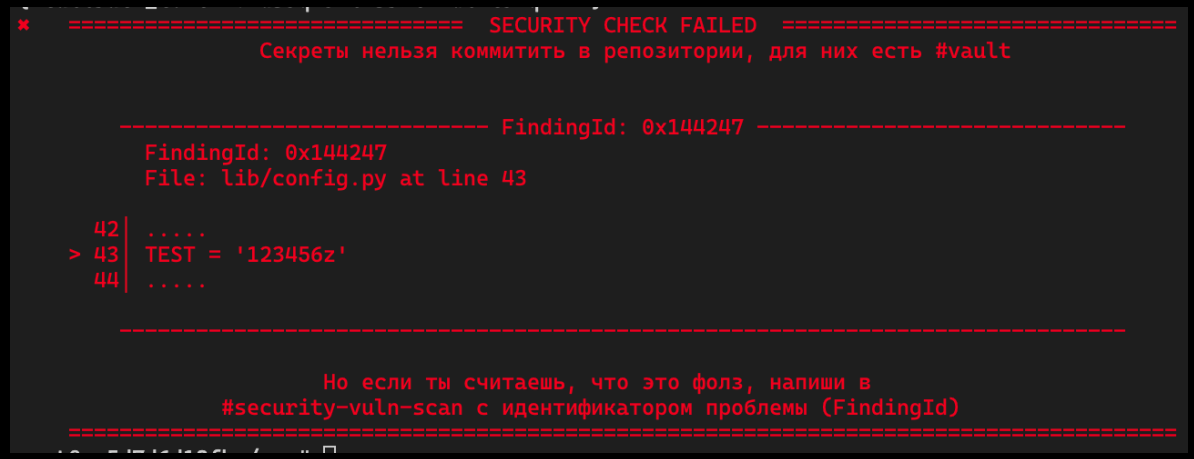
- **Hashicorp Vault** with self-service UI for devs – place a secret for your service yourself
- People tend to commit secrets (intentionally or accidentally) no matter how you teach them
- We scan every commit of new code, every new Docker image in a private registry

Avito’s Code Security Pipeline

Secrets Management | Processes

- Scanning on **pre-receive** stage prevents new secrets reaching origin (high-confidence findings)
- Secrets we miss during pre-receive stage are detected afterwards and resolved via general vuln management processes
- Credentials we are able to revoke – we revoke asap
- Tasks for Devs about changing compromised secrets are ranked higher
- Any old credentials are blocklisted, cannot be used as new credentials somewhere else

example of push being blocking during pre-receive



Problems of Finding Secrets in Code

So, the code

```
slack_token = 'xoxb-263594206564-FGqddMF8t08v8N7Qg
slack = SlackClient('xoxb-263594206564-FGqddMF8t
```

REGEX

```
priv = '''-----BEGIN RSA PRIVATE KEY-----
MIIBOgIBAAJBAKj34GkxFhD90vcNLYLInFEX6Ppy1tPf9Cnzj4p4WGeKLs1Pt8Qu
KUpRKfFLfRYC9AIKjbJTWit+CqvjWYzv
wEAAQJAIJLixBy2qpFoS4DSmoEm
o3qGy0t6z09AIJtH+5OeRV1be+N
Ba88vQENZiRm0GRq6a+HPGQMd2k
TQIhAKMSvzIBnni7ot/OSie2TmJ
rTQAevXysE2RbFDYdAiEBCUEaRQnMnbp7
9mxDXDf6AU0cN/RPBjb9qSHDcWZHGzUCIG2Es59z8ugGrDY+pxLQnwfotadxd+Uy
v/Ow5T0q5gIJAiEAyS4RaI9YG8EWx/2w0T67ZUVAw8eOMB6BIUg0Xcu+3okCIBOs
/5OiPgoTdSy7bcF9IGpSE8ZgGKzgYQVZeN97YE00
-----END RSA PRIVATE KEY-----'''
```

REGEX

```
tk = 'Skpv3aLWKafqX9ttj1aPJdGvz
```

ENTROPY!

```
MY'

url = 'https://test.com/item/Skpv3aLWKafqX9ttj1aPJdGvz9DVp7KJO1kMZ7MY'

login = {
    'user': 'bot',
    'pwd': 'GWHhIzyNeO'
}
```

```
.m
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return [self loadSomethingElseForTableView:tableView];
}
```

- typed secrets: detectable with regexes
- untyped secret: entropy to the rescue
- clear false positive with entropy
- not enough data for entropy
- names of methods has big entropy

More ways to detect. And a showstopper

```
login = {
    'user': 'bot',
    'pwd': 'GWHhIzyNeO'
}
```

By Plaintext

```
Secret to match:
GWHhIzyNe0
```

✓ `Regex: GWHhIzyNe0`

By hashed value

```
Secret to match:
386268c64ea12321f321cae069cf9a20489bc960
```

🤔 `What to hash in the code?!`

General questions to answer and problems to solve

- How to ignore keywords of a language?
- How to understand variables: names and values?
- How to take into account all cases of string init? (eq. single/double/triple quotes)

Should we buy a SAST?

Pro

- Excellent quality of language understanding **thanks to Abstract Syntax Tree**

Contra

- Low Extensibility: we need a separate SAST for every language we face
- Overkill, we need only AST
- We have to hack the internal machinery (give me a raw AST for the file)
- Does not work well with if we need to analyze a separate file without dependencies
- Expensive / some langs are not supported

Any ~~S~~AST builders?

Maybe there are libraries able to build an AST for us?

Pro

- Excellent quality of language understanding thanks to AST
- No need to hack SAST's machinery to get raw AST
- AST is pretty standardized

Contra

- Lack of libraries for Python (it's about building AST WITH Python for any other language)
- Lack of support
- Low Extensibility: we need a separate library for every language

Use native parser of a lang?!

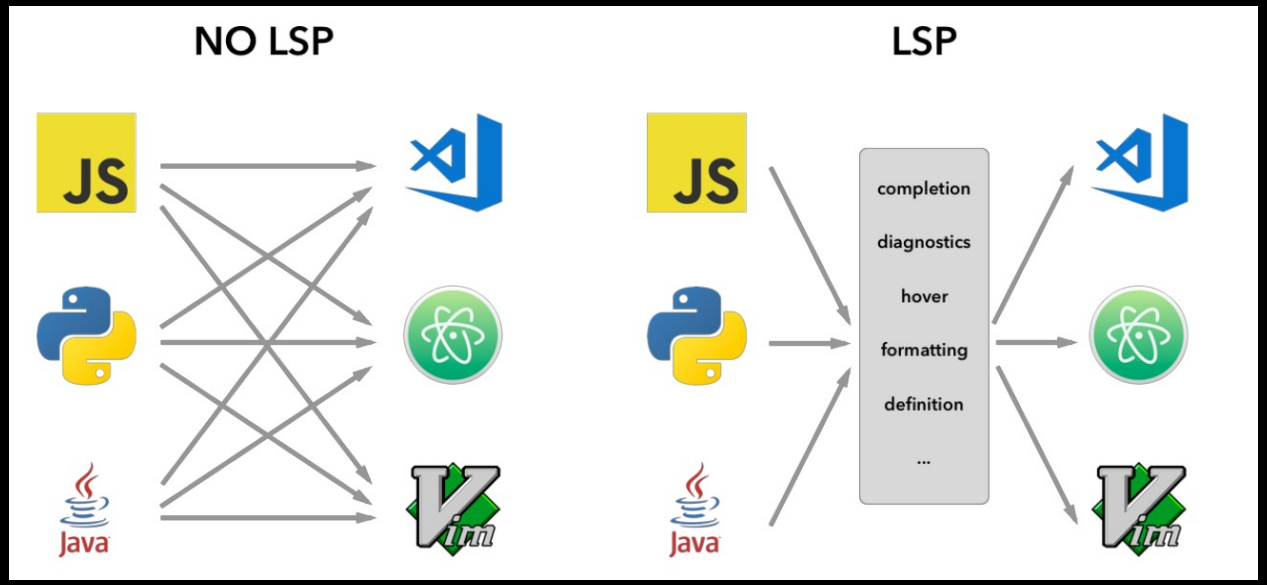
Let’s hack interpreters and compilers to intercept AST!
Let’s also keep all runtimes and compilers installed

LSP!

Language Server Protocol

What is it?

- Protocol used between an editor or IDE and a language server that provides language features like auto complete, go to definition, find all references etc.



- A Language Server is meant to provide the language-specific smarts and communicate with development tools over a protocol that enables inter-process communication.
- The idea behind the Language Server Protocol (LSP) is to standardize the protocol for how such servers and development tools communicate.

LSP!

Language Server Protocol

Semantic Tokens (🔁)

Since version 3.16.0

The request is sent from the client to the server to resolve semantic tokens for a given file. Semantic tokens are used to add additional color information to a file that depends on language specific symbol information. A semantic token request usually produces a large result. The protocol therefore supports encoding tokens with numbers. In addition optional support for deltas is available.

```
fmt.Printf("Hello, my name is %v!", name)
fmt.Printf("Hello")
```

"Hello,·my·name·is·%v!" 23 chars

language	go
standard token type	String
foreground	*#CE9178*
background	#1E1E1E
contrast ratio	6.31
semantic token type	string
modifiers	modification
foreground	string
	string meta.embedded.assembly
	{ "foreground": "#ce9178" }
textmate token	Hello,·my·name·is· (18)
textmate scopes	string.quoted.double.go source.go

OUTPUT TERMINAL DEB

```
": "fatal: not a git repos

epository for path='/Users/
de": 128,
orCode": "NotAGitRepository
mand": "rev-parse",
```

```
{
```

```
export enum SemanticTokenTypes {
    namespace = 'namespace',
    /**
     * Represents a generic type. Acts as a fallback for types which
     * can't be mapped to a specific type like class or enum.
     */
    type = 'type',
    class = 'class',
    enum = 'enum',
    interface = 'interface',
    struct = 'struct',
    typeParameter = 'typeParameter',
    parameter = 'parameter',
    variable = 'variable',
    property = 'property',
    enumMember = 'enumMember',
    event = 'event',
    function = 'function',
    method = 'method',
    macro = 'macro',
    keyword = 'keyword',
    modifier = 'modifier',
    comment = 'comment',
    string = 'string',
    number = 'number',
    regexp = 'regexp',
    operator = 'operator'
    /**
     * @since 3.17.0
     */
    decorator = 'decorator'
```

LSP! No ☹

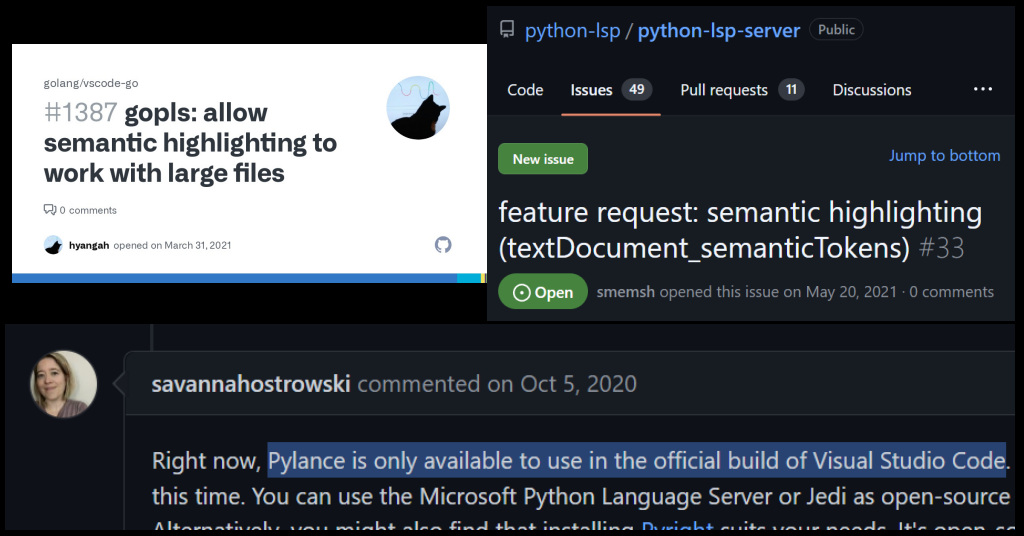
Language Server Protocol is still for future use

Pro

- Solves all our problems
- No need to hack anything to reach our needs
- Standard protocol, unification

Contra

- Language servers has lack of support for Semantic Tokens (python-lsp)
- Some language servers cannot be used standalone (Pylance)
- Performance issues (gopls)
- Low Extensibility: we need to run N language servers



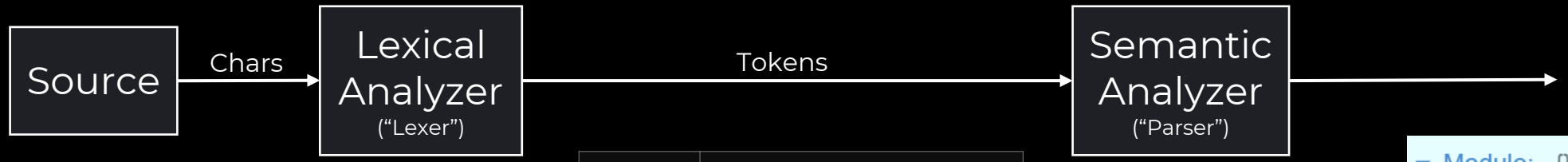
Quick reminder about the problem we're trying to solve ☺

We want to "understand" semantics of code to search secrets with maximum efficiency.

Understanding SAST

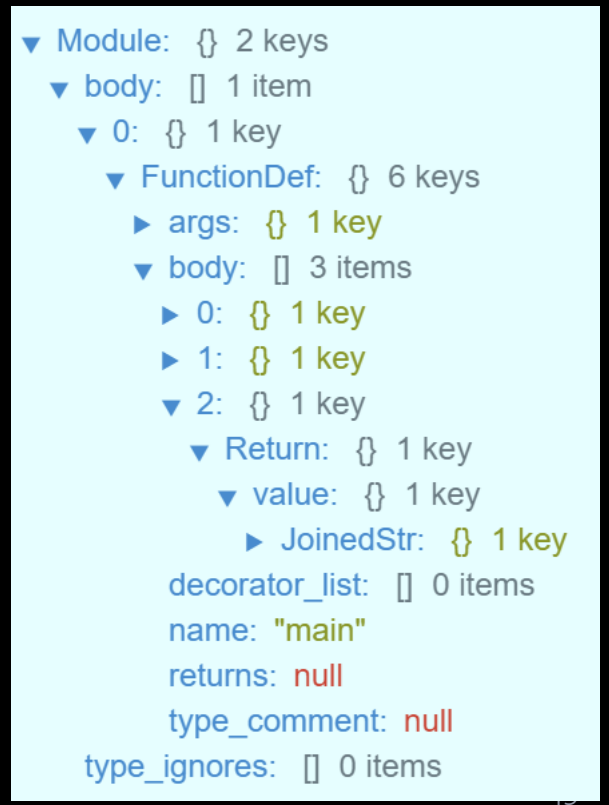
How SAST works (as well as compilers)

Let’s borrow the basics for our needs




```
def main(arg: str):
    a = 3
    b = 'hello'
    return f'{a}{b}'
```

def	meta.function
main	entity.name.function
(	punctuation.begin
arg	variable.parameter
:	punctuation.separator
str	class.typeHint
)	punctuation.end
:	punctuation.function.begin
a	variable.declaration
=	keyword.operator.assignment
3	constant.numeric.dec
b	variable.declaration
=	keyword.operator.assignment
'hello'	string.quoted.single
return	keyword.control.flow
f'{a}{b}'	string.quoted.single



Stage 1: Lexing

Did you mention on the previous slide?

```
def main(arg: str):
    a = 3
    b = 'hello'
    return f'{a}{b}'
```

Stage 1: Lexing

Did you mention on the previous slide?

```
def main(arg: str):
    a = 3
    b = 'hello'
    return f'{a}{b}'
```

Stage 1: Lexing

Syntax Highlighting! How does it work?!

No need to understand. It's FREE and FAST (Still regexes though)

```
def main(arg: str):
    a = 3
    b = 'hello'
    return f'{a}{b}'
```

Stage 1: Lexing via Syntax Highlighting

Knows how to highlight – read “tokenize” – 536 langs and formats

Open source and free

Able to output tokens raw!

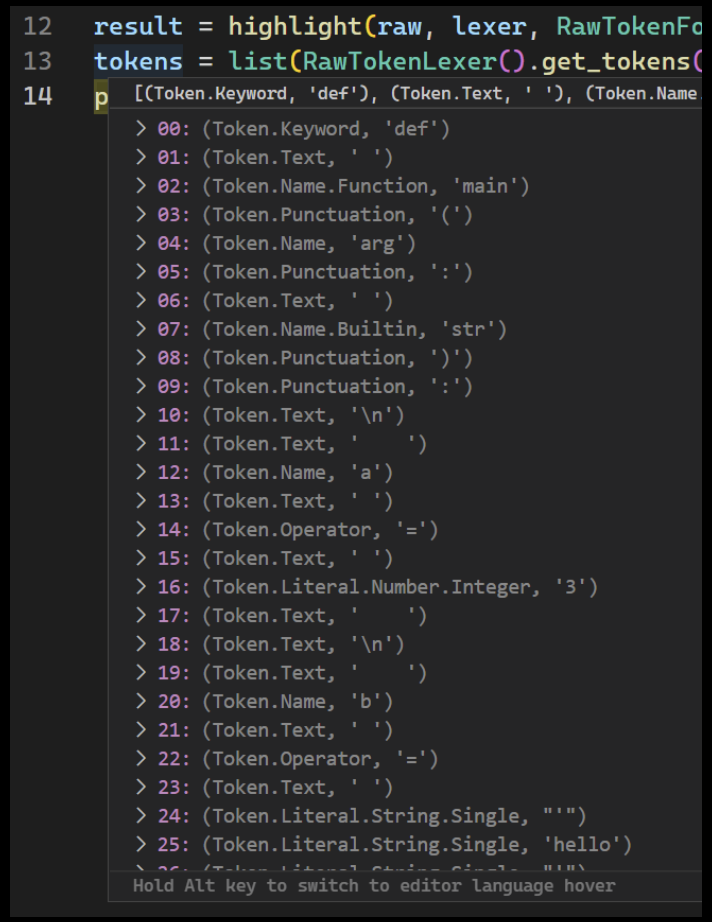
```
def main(arg: str):
    a = 3
    b = 'hello'
    return f'{a}{b}'
```



```
12  result = highlight(raw, lexer, RawTokenFo
13  tokens = list(RawTokenLexer().get_tokens(
14  p  [(Token.Keyword, 'def'), (Token.Text, ' '), (Token.Name.
     > 00: (Token.Keyword, 'def')
     > 01: (Token.Text, ' ')
     > 02: (Token.Name.Function, 'main')
     > 03: (Token.Punctuation, '(')
     > 04: (Token.Name, 'arg')
     > 05: (Token.Punctuation, ':')
     > 06: (Token.Text, ' ')
     > 07: (Token.Name.Builtin, 'str')
     > 08: (Token.Punctuation, ')')
     > 09: (Token.Punctuation, ':')
     > 10: (Token.Text, '\n')
     > 11: (Token.Text, '    ')
     > 12: (Token.Name, 'a')
     > 13: (Token.Text, ' ')
     > 14: (Token.Operator, '=')
     > 15: (Token.Text, ' ')
     > 16: (Token.Literal.Number.Integer, '3')
     > 17: (Token.Text, '    ')
     > 18: (Token.Text, '\n')
     > 19: (Token.Text, '    ')
     > 20: (Token.Name, 'b')
     > 21: (Token.Text, ' ')
     > 22: (Token.Operator, '=')
     > 23: (Token.Text, ' ')
     > 24: (Token.Literal.String.Single, "'")
     > 25: (Token.Literal.String.Single, 'hello')
     Hold Alt key to switch to editor language hover
```

Stage 1: Lexing via Syntax Highlighting

What do we get with it?



Benefits

- Tokenization with understanding of language
- Types of tokens – just ignore useless ones

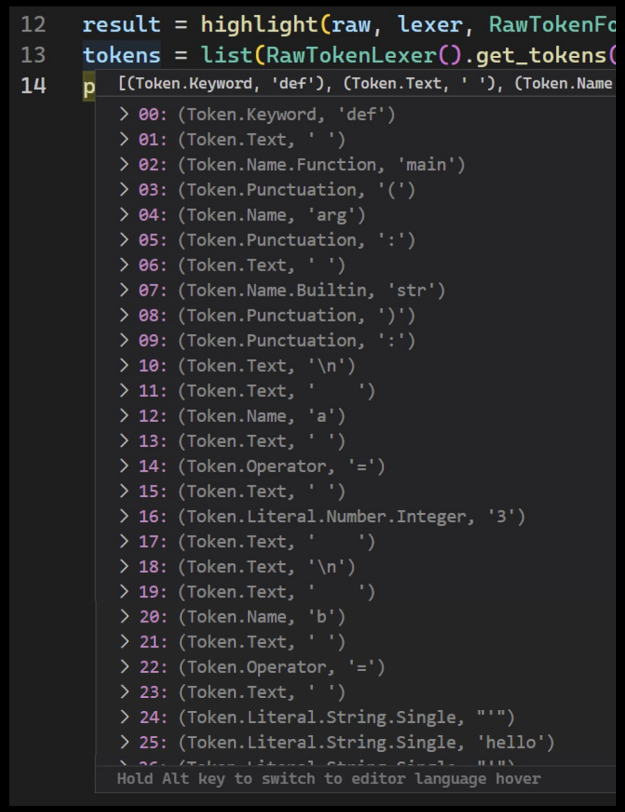
Still unresolved issues

- No info about boundaries of tokens – you do it
- Variables are still not detected – task for stage 2

Stage 2: True AST for variable detection or.. ?

Further research

- We need to create rules for variable detection without recreating regex engine
- Research showed the following:
  - **Token type pattern** matters – place for regex
  - Token values are auxiliary



How do we match this ‘token type pattern’?

Let’s convert Token’s type to a single char and concat, eq.:

```
Token.Keyword = ‘k’
Token.Text = ‘t’
Token.Name.Function = ‘f’
…
```

Stage 2: True AST for variable detection or.. ?

Further research

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Token Type	k	t	n	p	n	p	t	b	p	p	t	t	n	t	o	t	i	t	t	t	n	t	o	t	s	s	s	t	…
Token Value	def		main	(	arg	:		str	)	:	\n		a		=		3		\n		b		=		‘	hello	‘	\n	…

Stage 2: True AST for variable detection or.. ?

Further research

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Token Type	k	t	n	p	n	p	t	b	p	p	t	t	n	t	o	t	i	t	t	t	n	t	o	t	s	s	s	t	…
Token Value	def		main	(	arg	:		str	)	:	\n		a		=		3		\n		b		=		‘	hello	‘	\n	…

- Rule
  - StreamPattern: (n)t*(o|p)t*(s)(s)(s)t
  - MatchRules:
    - 2: “=“
    - 3: [‘””, ‘\’]
    - 5: [‘””, ‘\’]
  - MatchSemantics:
    - 1: name
    - 4: value

Stage 2: True AST for variable detection or.. ?

Further research

var: **b**

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Token Type	k	t	n	p	n	p	t	b	p	p	t	t	n	t	o	t	i	t	t	t	n	t	o	t	s	s	s	t	…
Token Value	def		main	(	arg	:		str	)	:	\n		a		=		3		\n		b		=		‘	hello	‘	\n	…

- Rule
  - StreamPattern: `(n)t*(o|p)t*(s)(s)(s)t`
  - MatchRules:
    - 2: “=“
    - 3: [‘”’, ‘\’]
    - 5: [‘”’, ‘\’]
  - MatchSemantics:
    - 1: name
    - 4: value

ktnpnptbppttntotitttntotssstpnptbppt

Match 1	20-28	ntotssst
Group 1	20-21	n
Group 2	22-23	o
Group 3	24-25	s
Group 4	25-26	s
Group 5	26-27	s

Outcomes

Enabler
for deeper analysis
(eq. “suspicious variable names + high entropy variable value”)

+40%
performance boost
Lower amount of analyzed strings /entropy calculations / etc.

+30%
more findings
on Avito’s codebase

Limitations

- Plaintext files without any semantics are obviously not covered
- Var Detection Rules (VDRs) are language-specific (simpler than building a SAST though)
- Rules does not guarantee full coverage, deep testing required
- Hard (but possible) to detect when secrets concat from innocent parts

Full Picture

Architecture of the service

File Model

- Full content
- Line start-end positions
- Line content caches
- Tokens

Splits file into tokens

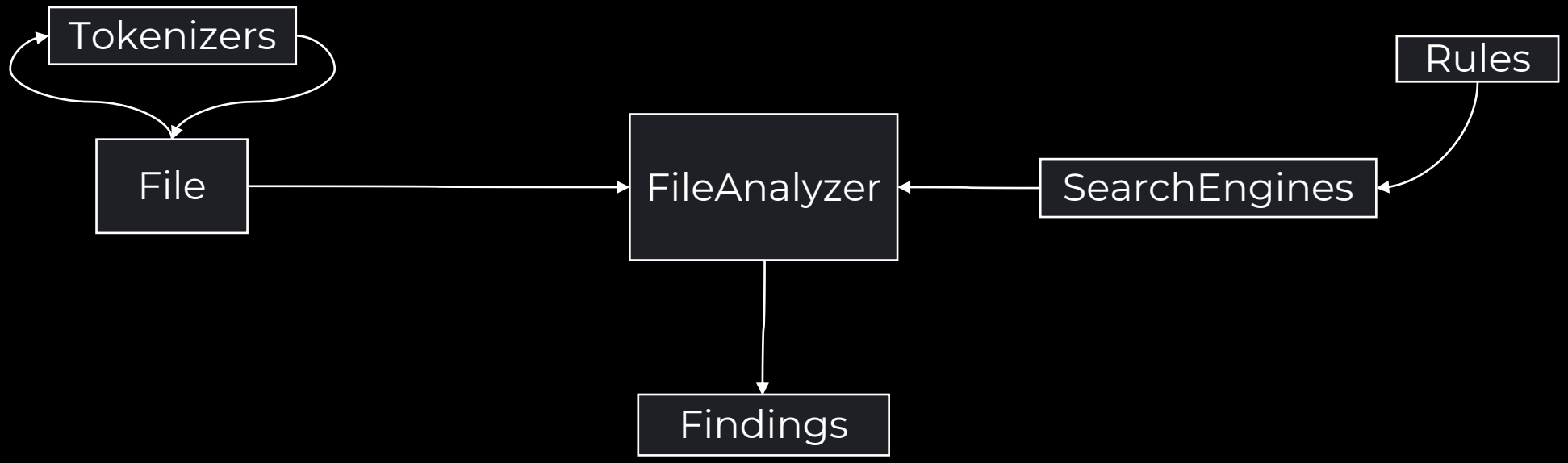
Tokenizer

FullContentTokenizer
PerWordTokenizer
LexerTokenizer with VDRs

Inspects a token

SearchEngine

RegexEngine
EntropyEngine
HashedValueEngine



Full Picture

Open Source?

Later this year

Thanks

Stay Secure!

ntoskrnl

khechumov