A Method to Enhance the Security Capability of Python IDE

Vinh Pham[1], Namuk Kim[2], Eunil Seo[2], Jun Suk Ha[3], and Tai-Myoung Chung[1](✉)

[1] Department of Computer Science and Engineering, Sungkyunkwan University, Suwon, Korea
vinhpham@g.skku.edu, tmchung@skku.edu

[2] Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon, Korea
{nukim8275,seoei2}@g.skku.edu

[3] Department of Mathematics and Computer Science, University of Illinois at Urbana-Champaign, Urbana, USA
junsukh2@illinois.edu

Abstract. The majority of applications running on the Internet are web applications; however, these applications are vulnerable to arbitrary code execution and database manipulation by Cross-Site Scripting or SQL injection attacks. The fundamental reason of these vulnerabilities is that web applications use a string type for assembling heterogeneous computer languages' syntax for a particular language. To cope with these vulnerabilities, we propose a language-based scheme, in which the programming language itself provides security capabilities by a method of the syntax embedded in Python. Furthermore, the proposed solution supports backward compatibility and higher portability to other languages as well as Python. To improve the debugging difficulty caused by a language-based scheme, we propose a trace-processor that has post-mortem debug ability. We implement the proposed solution as a development environment, named Python-S, based on CPython's source code. Python-S successfully displays the protection capabilities for the SQL injection attack.

Keywords: Code injection · Python · Web application · Programming language

1 Introduction

Web applications are frequently targeted by attackers due to their ease of use, omnipresence, demand, and growing number of users. Consequently, the number of security vulnerabilities reported in connection with web applications has been steadily increasing, in parallel to the growing significance of a web application paradigm. According to OWASP [1], SANS Institute [2], and Trustwave [3], code injection vulnerabilities are the most potent vulnerabilities that threaten

© Springer Nature Switzerland AG 2019
T. K. Dang et al. (Eds.): FDSE 2019, LNCS 11814, pp. 399–410, 2019.
https://doi.org/10.1007/978-3-030-35653-8_27

the security of web applications. As a result, web application security has started to attract more attention than ever before from both academia and industry.

OWASP Top 10 Most Critical Web Application Security Risks [1] ranks two code injection vulnerabilities second and seventh:

– **SQL Injection (SQLi):** is a web security vulnerability in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands. A successful SQL injection exploit can read sensitive data, modify data (Insert/Update/Delete) or execute administrative operations on the database [1].
– **Cross-site Scripting (XSS):** is a web security vulnerability in which malicious scripts are injected into otherwise benign and trusted websites. XSS occurs when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user [1].

This paper will examines the most basic web application model, "One Web Server, One Database", as well as a web application in the form of CGI (Common Gateway Interface) Scripts. An HTTP server can be setup in a way that whenever a file from a certain directory is requested, that file is not sent back. Instead, it is executed as a program. The program's outputs are sent back to the user's browser to be displayed. This configuration of HTTP server is called the Common Gateway Interface and the programs (more specifically, the requested file) are called the CGI scripts [4].

Web applications usually employ different heterogeneous computer languages. In the rest of this paper, we will use the following naming conventions:

– **Hosting Language:** The language that was used to program the actual application. In our paper, we define Python as a hosting language.

– **Foreign Language:** All other computer languages that are used within the application. (e.g., SQL, HTML, and JavaScript)

According to Stack Overflow's annual Developer Survey 2019 [5] and JetBrains Python Developers Survey 2018 [6], Python is the fastest-growing programming language today, with 52% of respondents stating that they use Python for web development. It is necessary to have a solution for the injection problems of web applications that is written in Python language.

Based on the above observation, we design and implement a development environment, the so-called Python-S. Python-S has language-based security features, that prohibit developers from writing source code involuntarily and implicitly assembling foreign language code from attacker-provided malicious string values. In this paper, our contributions are as follows:

– We present an additional security syntax that helps protect Python web application against SQLi and XSS attacks.
– We implement Python-S compiler and interpreter based on CPython.
– We verify the protection capabilities of Python-S to prevent the SQL injection attack.

– We propose the trace-processor obtaining post-mortem debugging ability, which becomes a component of an enhanced IDE of Python language.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 explains the design principle of our proposed solution. Section 4 describes how we implemented Python-S based on the original CPython's source code and components. Section 5 justifies Python-S against a concrete SQLi attack scenario. Section 6 provides the Discussion while Sect. 7 concludes the paper.

2 Related Work

Although extensive research has been conducted and a large amount of effort has been spent over the decades, code injection attacks against web applications are still prevalent. Some notable academic work has been published in the field of security of web applications as follows. Our approach can be categorized into *Secure programming* category.

– *Secure programming:* Juillerat [7] designed a Java library (Stone) that can enforce security to create robust, secure web applications and manage the vulnerabilities of SQLi and XSS. Grabowski et al. [8] developed a type system for Java language to enforce secure programming guidelines to the prevention of XSS attacks.
– *Vulnerability detection (or prediction) & prevention:* Kals et al. [9] developed a black-box vulnerability scanner (named Secubat) with the ability to identify SQL and XSS. Shar and Tan [10] developed a prototype PHPMinerI that predicts SQLI and XSS by employing machine-learning techniques on input sanitization patterns.
– *Attack detection (or prediction) & prevention:* Su and Wassermann [11] proposed SQLCHECK that prevents SQLi attacks by context-free grammar and parsing techniques. XSSDS [12] is a proxy-based system, which intercepts and compares the HTTP requests and responses to detect XSS attacks.

Up until the time writing this paper, to the best of our knowledge, Python-S is the first security-oriented initial IDE for Python. In term of Python's security capability, more specifically, the programming language extensibility approach for enhancing it's security capability, there are some latest research that we consider as closely related to our work and will be mentioned here as follow.

Fulton et al. introduced *regular string types* [13] which employed *regular expression* to statically verify that sanitization for user input has been performed correctly. This is equivalent to our work in the way that authors also objected using string type in web application programming because it's risk of vulnerabilities. However, they just implemented their work as a Python library in order to prove their hypothesis but not an complete solution for Python.

PyT [14] and Pythia [15] are two tools that detecting web application vulnerabilities for Flask and Django web programming framework, respectively. They might be equivalent with our proposed solution in the way that they also employed Python's primitive components (package) or modified Python's interpreter in order to achieve their target. But our Python-S is different from them in term that not only Python-S isn't binded to any specific web framework but also Python-S is a code injection vulnerability mitigation solution, not vulnerabilities detecting solution. More over, by implementing as an IDE, Python-S can be integrated more techniques into and evolved in the future in order to solve other security problems.

3 Design Principle

3.1 The String Type's Defect

Usually, while programming in Python, a foreign language code (e.g., a SQL command) was assembled by concatenating the *data* (provided by the user, but it could be by an attacker as well) as a variable, at the end of a pre-defined string, which represents a portion of SQL *code*.

```
sql = "SELECT * FROM customer WHERE userid = ’%s’"%uid
```

In most cases, this method works well. However, when the *data* is malformed and combined with the *code*, the final results end up having a completely different meaning or expression from what the programmer intended.

The programmer's intent is that the *data* must not carry any code meaning, just simply the value. But an external entity's (such as a database management system or web browser) interpreter (parser) can't distinguish between the *code* and the *data*, because they are serialized as a whole single string.

The fault is in our string-based method of assembling foreign language code, because it doesn't provide any capability for strictly separating *code* and *data*. Hence, an attacker can exploit this discord in the respective view of the programmer and application to conduct a code injection attack, as in Fig. 1. Because external interpreters have no knowledge of the programmer's intent, they just simply parse the dispatched foreign language code according to their language grammar. By providing malformed data, the attacker can cause the external interpreter viewing dispatched code with different code's meaning from the programmer's intent, as in Fig. 2.

$$\underbrace{SELECT\ *\ FROM\ customer\ WHERE\ userid\ =}_{Code}\ \underbrace{uid}_{Data}$$

$$\underbrace{SELECT\ *\ FROM\ customer\ WHERE\ userid\ = uid}_{String}$$

Fig. 1. The discord in the respective view of programmer and application

$$\underbrace{SELECT}_{Code}\underbrace{*}_{Data}\underbrace{FROM}_{Code}\underbrace{customer}_{Data}\underbrace{WHERE}_{Code}\underbrace{userid}_{Data}\underbrace{=}_{Code}\underbrace{\underbrace{X}_{Data}\underbrace{OR}_{Code}\underbrace{'1'}_{Data}\underbrace{=}_{Code}\underbrace{'1'}_{Data}}_{Malformed\ Data}$$

Fig. 2. SQLi attack by providing malformed data

3.2 Language-Base Scheme Concept

We employ the approach introduced in the research by Jonh et al. [16] to secure Python as a hosting language. This research proposes a language-based concept of assembly foreign language syntax that provides a robust security guarantee. The concept is constructed on three key elements: **Datatype**, **Language integration**, and **External interface**.

By employing a new **datatype** (replacement of string type) to encapsulate foreign language code, we can tokenize the code. This helps enforce explicit code creation and strict separation of *code* and *data* (enforcement that all creation of foreign language code must be explicit by the hosting language's capabilities - **language integration**).

For example, when encapsulated in the new **datatype**, the SQL command at Sect. 3.1 can be represented as follows:

```
sql = { select-token, meta-char(*), from-token,
        tablename-token(customer), where-token,
        fieldname-token(userid), metachar(=), metachar(’),
        stringliteral(uid’s value), metachar(’) }
```

In order to execute a successful code injection attack, the attacker must cause the hosting language application to dynamically includes parts of his malicious values into a new datatype, explicitly in a syntactical code context, such as tokens of code, or literal data. At this point, only data (raw string or number without syntactic meaning) are allowed to be added to the new datatype. Consequently, the developer is not able to write source code that involuntarily and implicitly creates embedded code from the attacker-provided string value. Hence, string-based code injection is impossible [17].

The **External interface** plays a role as a mediate abstract layer between the hosting language application and other external entity interpreters. This layer provides secured code serialization operation that will translate foreign language codes encapsulated in the above specific-purpose **datatype** into a character-based representation, because most of external entities, such as SQL DBMS or web browser's accepted HTTP responding, still prefer string-based type as their communication tool.

3.3 Backward-Compatible

In order to get the old Python source code backward-compatible with our solution, we employed the "Mark-up Signifier" programming procedure of

Johns et al. [18], which is described in Sect. 4.1. With just a few syntax modifications to the source code, previously written Python web applications can obtain code injection resistance provided by our solution. This also helps the programmer to easily and comfortably adapt to our solution while assembling foreign language code within Python source code.

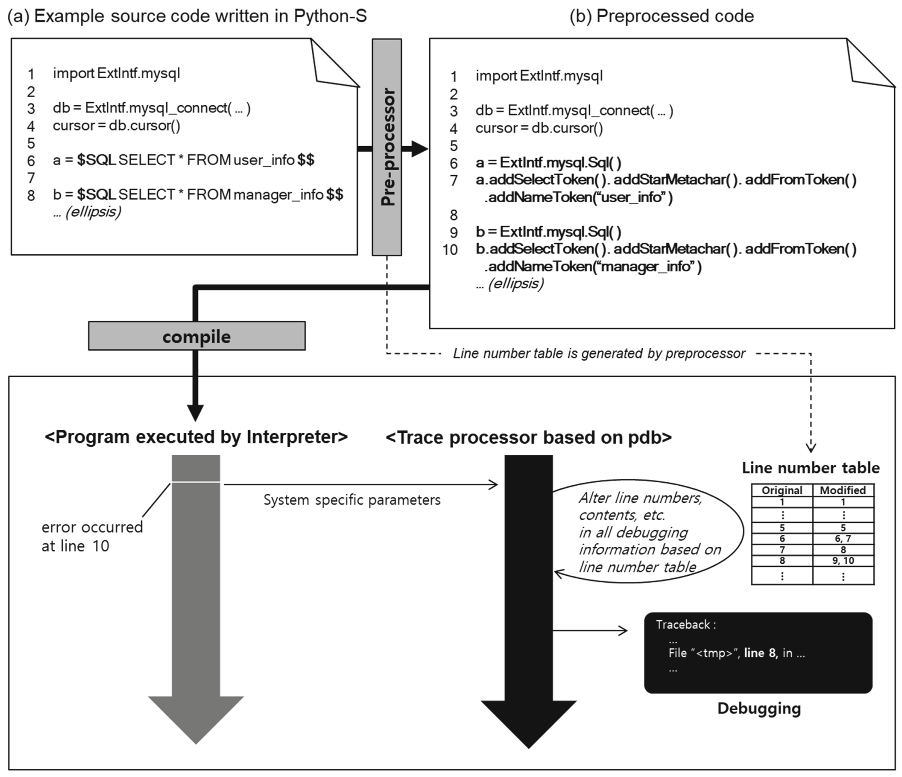
3.4 Trace-Processor

Johns et al. [18] has the drawback that the code written by the programmer is different from the pre-processed code that is actually interpreted. There is a chance that error occurs within the code region that was processed by the pre-processor (which is explained in Sect. 4.1). When debugging these errors, the debugger will refer to lines of code that are unknown to the programmer.

Taking into consideration the fact that most contemporary web applications were developed in the Event-driven design paradigm, traditional debug techniques (e.g. trial and error, setting break-point or print-out of the variable's value) can't cover all execution paths, which may lead to errors within the pre-processed code region. Moreover, by the web application's nature, it will be an advantage if web application can be fault-tolerant. That means that rather than stopping when error occurs, the web application should continue running and handling errors by an exception handler (such as Python "*try - except*" procedure), as well as allow the programmer to debug after an error has occurred.

For this reason, we propose additionally using a trace-processor that runs alongside the hosting language application. We constructed the trace-processor based on the Python's Debugger (*pdb*), which has already obtained post-mortem debugging ability. When an error occurs (an exception is raised), the hosting language application will use the Python sys.exc_info() function to catch the Python interpreter's system specific parameters (e.g., *sys.last_traceback*, *sys.last_value*, *sys.last_type*), and will then continue running or handling the exception. Later, the trace-processor can use these system specific parameters in order to reconstruct the Python interpreter's call stack, up until the point where the exception was raised. The call stack contains frames that are essentially a snapshot of the execution of hosting language web application at the moment that an error occurs. Hence, the programmer can inspect all necessary information (e.g. line of error code, local variables in each function up until the exception), to locate the error.

Suppose a Python program is written using a markup signifier as shown in Fig. 3(a). After this source code is pre-processed, the code will be formed as shown in Fig. 3(b). As mentioned in Sect. 4.1, pre-processed code is different from the original code. To deal with the difference between the original and pre-processed code during debugging, a line number table is generated that stores the line numbers of modified code, which correspond to each line number of the original code being stored.


(a) Example source code written in Python-S

```
1   import ExtIntf.mysql
2
3   db = ExtIntf.mysql_connect( ... )
4   cursor = db.cursor()
5
6   a = $SQL SELECT * FROM user_info $$
7
8   b = $SQL SELECT * FROM manager_info $$
    ... (ellipsis)
```

(b) Preprocessed code

```
1   import ExtIntf.mysql
2
3   db = ExtIntf.mysql_connect( ... )
4   cursor = db.cursor()
5
6   a = ExtIntf.mysql.Sql( )
7   a.addSelectToken( ). addStarMetachar( ). addFromToken( )
      .addNameToken("user_info" )
8
9   b = ExtIntf.mysql.Sql( )
10  b.addSelectToken( ). addStarMetachar( ). addFromToken( )
      .addNameToken("manager_info" )
    ... (ellipsis)
```

Line number table

Original	Modified
1	1
⋮	⋮
5	5
6	6, 7
7	8
8	9, 10
⋮	⋮

```
Traceback :
  ...
  File "<tmp>", line 8, in ...
  ...
```

Pre-processor
compile
Line number table is generated by preprocessor
<Program executed by Interpreter>
<Trace processor based on pdb>
error occurred at line 10
System specific parameters
Alter line numbers, contents, etc. in all debugging information based on line number table
Debugging

(c) Trace processor's operation process when error occurred

Fig. 3. Trace processor's operation process

While pre-processed code is executed by the interpreter, suppose an error occurs at line 10. Then system specific parameters are delivered to the trace-processor, and it reconstructs the call stack to trace the source of the problem. When printing debugging information, line numbers, and contents are altered by referencing the line number table, so that the programmer can understand the information (see Fig. 3(c)).

4 Implementation

According to our design principle, we implemented three components and a trace-processor into Python; furthermore, we modified CPython to operate with the secured Python source code that has new syntax. Figure 4 describes the overall architecture of Python-S with the following components.

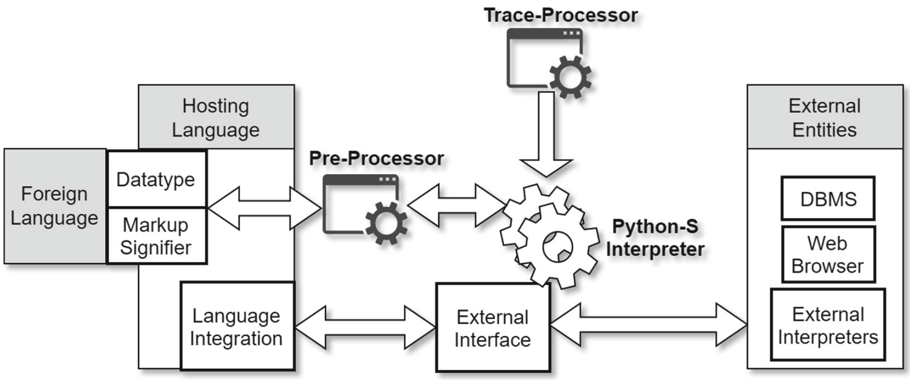


Fig. 4. Python-S architecture

4.1 Mark-Up Signifier and Pre-processor

Instead of using string type to assemble foreign language code as usual, Python-S requires the programmer must to explicitly state his or her intent when creating or assembling foreign language code in the hosting language source code by the "Mark-up Signifier" programming procedure in Fig. 4.

Foreign language codes, such as SQL, HTML, and JavaScript will be integrated with the hosting language (e.g., Python) source code, and framed by an explicit mark-up signifier. The Mark-up Signifier will help the pre-processor differentiate which language the followed code belongs to. We propose the following mark-up signifier for each foreign language, as follows:

```
\\ Mark-up Signifier for SQL language
s = $SQL SELECT * FROM $$
\\ Mark-up Signifier for HTML language
h = $HTML <p> This is a paragraph.</p> $$
\\ Mark-up Signifier for JavaScript language
j = $JS var d = new Date(); $$
```

A source-to-source pre-processor completely scans Python source code, before it is compiled by the Python's compiler. The pre-processor identifies the mark-up signifiers, and automatically adds API-calls that corresponds to the foreign language that the mark-up signifier represents.

```
\\ s = $SQL SELECT * FROM $$
s.addSelectToken().addStarMetachar().addFromToken()
```

4.2 Language Integration Through API

Python language must be upgraded in order to co-operate with the pre-processor. To obtain this target, we design the new datatype in the form of a Python programming library (API). This API will provide an interface that will be used by a pre-processor. The API contains methods that are called by the pre-processor to tokenize foreign language code into three token classes (code-token, identifier-token, and data-token) depending on their language syntactic characteristics. When those API's methods are called, they create a token, and add it to the new datatype variable.

4.3 External Interface as Abstract Layer

We design the External Interface, which is a centralized instance that contains the domain-specific knowledge about the respective foreign languages, in the form of a Python module, as in Fig. 4. In order to prevent potential code injection risks, programmers are forced to communicate with external entities, such as querying to a database, or responding to an HTTP request, only via this specific module. We present the usage of the External Interface in the case of HTML responding and SQL querying, as follows:

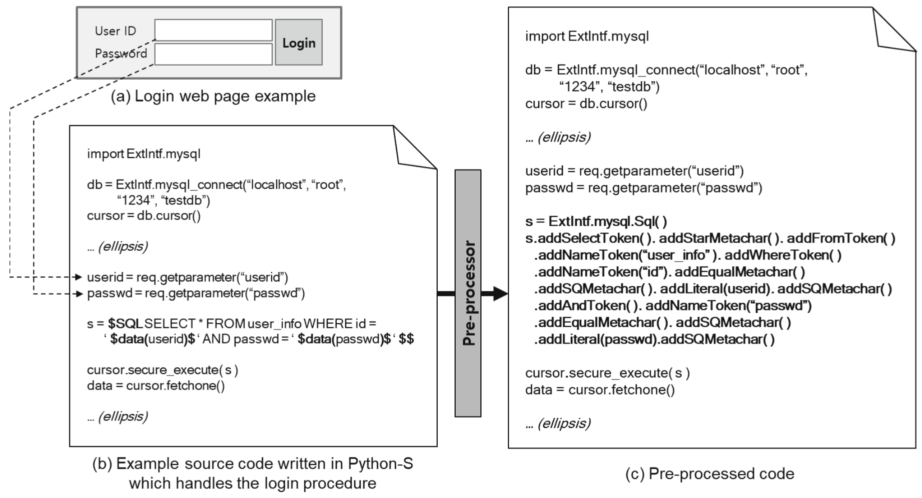
```
import ExtIntf \\ Import External Interface module
ExtIntf.secure_htmlrespond(h) \\ Respond to and HTML request
\\ Query SQL database
ExtIntf.secure_connect("localhost","testuser","test123","TESTDB")
ExtIntf.secure_execute(s)
```

5 Evaluation

In this section, we describe how Python-S works through a simple example of a log-in web page, which receives a username and password from a user, and passes a query to the MySQL database to verify the user information. The procedures of Python pre-processing are as follows: Fig. 5(a) shows how the log-in page is configured. Figure 5(b) shows that the Python source code for handling the log-in procedure can be written using markup signifiers. Figure 5(c) shows that after the source code is pre-processed, the code contains script-specific object generation script and API calls for each SQL command token.

The defense procedures for SQL injection attacks are as follows: Fig. 6(a) shows that a malicious user enters the code for SQL injection attack. Figure 6(b) shows that through the pre-processed code, an SQL object is generated, and APIs or methods are called sequentially according to the SQL command token. Each call appends the query string to S.Sql_String, and inqueue according to the type value to S.TokenType_Queue. When the successive 'add$X$' API calls are completed, S.Sql_String and S.TokenType_Queue contains data as shown in Fig. 6(c).

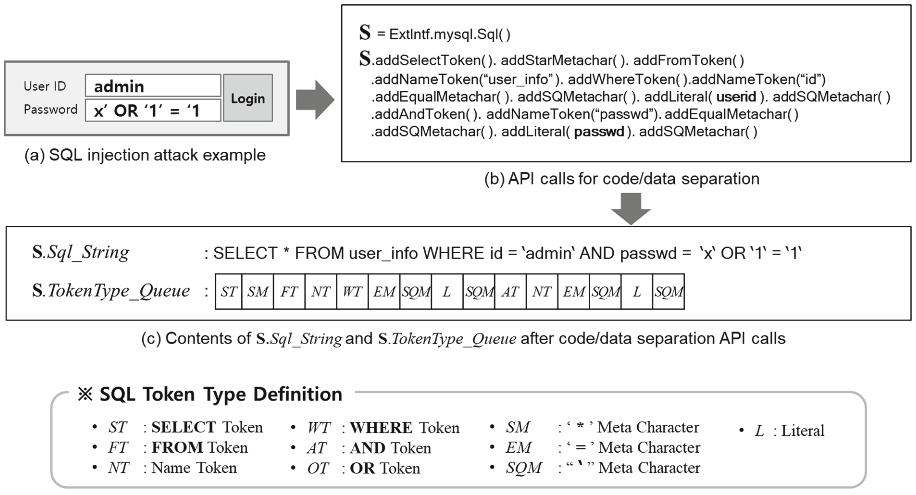
Finally, log-in handling code calls secure_execute method with argument S, and in the method, checkSqlDeformation method checks if there is potential SQL injection attack code in S in Fig. 7. In this example, due to malicious code injection, the contents of ActualTokenType_Queue are different from S.TokenType_Queue. Therefore, checkSqlDeformation method returns FALSE, and the SQL command is not executed.


User ID
Password
Login
(a) Login web page example
Pre-processor
(b) Example source code written in Python-S which handles the login procedure
(c) Pre-processed code

```
import ExtIntf.mysql

db = ExtIntf.mysql_connect("localhost", "root",
      "1234", "testdb")
cursor = db.cursor()

... (ellipsis)

userid = req.getparameter("userid")
passwd = req.getparameter("passwd")

s = $SQL SELECT * FROM user_info WHERE id =
    ' $data(userid)$ ' AND passwd = ' $data(passwd)$ ' $$

cursor.secure_execute( s )
data = cursor.fetchone()

... (ellipsis)
```

```
import ExtIntf.mysql

db = ExtIntf.mysql_connect("localhost", "root",
      "1234", "testdb")
cursor = db.cursor()

... (ellipsis)

userid = req.getparameter("userid")
passwd = req.getparameter("passwd")

s = ExtIntf.mysql.Sql( )
s.addSelectToken( ). addStarMetachar( ). addFromToken( )
  .addNameToken("user_info" ). addWhereToken( )
  .addNameToken("id"). addEqualMetachar( )
  .addSQMetachar( ). addLiteral(userid). addSQMetachar( )
  .addAndToken( ). addNameToken("passwd")
  .addEqualMetachar( ). addSQMetachar( )
  .addLiteral(passwd).addSQMetachar( )

cursor.secure_execute( s )
data = cursor.fetchone()

... (ellipsis)
```

Fig. 5. SQL injection attack prevention example - 1) Python source code pre-processing


User ID
admin
Password
x' OR '1' = '1
Login
(a) SQL injection attack example
(b) API calls for code/data separation
(c) Contents of S.Sql_String and S.TokenType_Queue after code/data separation API calls
※ SQL Token Type Definition

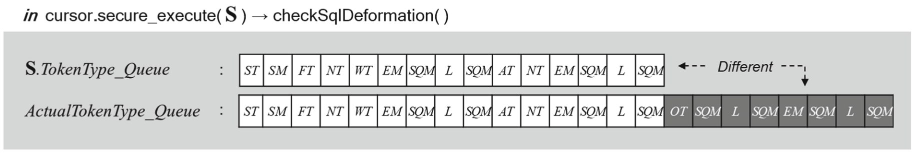
```
S = ExtIntf.mysql.Sql( )
S.addSelectToken( ). addStarMetachar( ). addFromToken( )
  .addNameToken("user_info"). addWhereToken( ).addNameToken("id")
  .addEqualMetachar( ). addSQMetachar( ). addLiteral( userid ). addSQMetachar( )
  .addAndToken( ). addNameToken("passwd"). addEqualMetachar( )
  .addSQMetachar( ). addLiteral( passwd ). addSQMetachar( )
```

S.*Sql_String* : SELECT * FROM user_info WHERE id = 'admin' AND passwd = 'x' OR '1' = '1'

S.*TokenType_Queue* : *ST* | *SM* | *FT* | *NT* | *WT* | *EM* | *SQM* | *L* | *SQM* | *AT* | *NT* | *EM* | *SQM* | *L* | *SQM*

• *ST* : **SELECT** Token	• *WT* : **WHERE** Token	• *SM* : ' * ' Meta Character	• *L* : Literal
• *FT* : **FROM** Token	• *AT* : **AND** Token	• *EM* : ' = ' Meta Character	
• *NT* : Name Token	• *OT* : **OR** Token	• *SQM* : " ' " Meta Character	

Fig. 6. SQL injection attack prevention example - 2) SQL injection attack attempt



S.*TokenType_Queue* : *ST* | *SM* | *FT* | *NT* | *WT* | *EM* | *SQM* | *L* | *SQM* | *AT* | *NT* | *EM* | *SQM* | *L* | *SQM*

ActualTokenType_Queue : *ST* | *SM* | *FT* | *NT* | *WT* | *EM* | *SQM* | *L* | *SQM* | *AT* | *NT* | *EM* | *SQM* | *L* | *SQM* | *OT* | *SQM* | *L* | *SQM* | *EM* | *SQM* | *L* | *SQM*

Fig. 7. SQL injection attack prevention example - 3) Detection and Prevention

6 Discussion

Usually, Python web application development employs not only Python's primitive component, but also it's abundant extensions in terms of library. Most Python extensions that require heavy calculation or raw performance (e.g., image processing) are still based on C language, because of it's effectiveness by nature as a low-level language. Although an application written in Python is already a memory safe program, there is still a possibility that a running Python program catches a memory error when it employs such C extension. In the future, we plan to extend Python-S to mitigate memory error (e.g., buffer overflow) for Python-S applications.

7 Conclusion

In this paper, we propose and implement Python-S, which is a security enhanced IDE of Python. Web applications developed by Python-S have security capabilities to prevent code injection attacks, such as SQLi and XSS. Python-S supports backward-compatibility for Python source code. We demonstrate that Python-S web application successfully defends against an SQL injection attack; furthermore, Python-S will foster secure web applications.

References

1. OWASP Homepage. https://www.owasp.org
2. Cwe/sans top 25 most dangerous software errors (2011). http://www.sans.org/top25-software-errors/
3. 2011 Trustwave Global Security Report. https://www.trustwave.com
4. Python 3 - CGI Programming. https://www.tutorialspoint.com
5. Stack Overflow's annual Developer Survey (2019). https://insights.stackoverflow.com/survey/2019#most-popular-technologies
6. JetBrains Python Developers Survey (2018). https://www.jetbrains.com/research/python-developers-survey-2018/
7. Juillerat, N.: Enforcing code security in database web applications using libraries and object models. In: Proceedings of the 2007 Symposium on Library-Centric Software Design, pp. 31–41. ACM (2007)
8. Grabowski, R., Hofmann, M., Li, K.: Type-based enforcement of secure programming guidelines — code injection prevention at SAP. In: Barthe, G., Datta, A., Etalle, S. (eds.) FAST 2011. LNCS, vol. 7140, pp. 182–197. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29420-4_12
9. Kals, S., Kirda, E., Kruegel, C., Jovanovic, N.: SecuBat: a web vulnerability scanner. In: Proceedings of the 15th International Conference on World Wide Web, pp. 247–256. ACM (2006)
10. Shar, L.K., Tan, H.B.K.: Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns. Inf. Softw. Technol. **55**(10), 1767–1780 (2013)
11. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. ACM SIGPLAN Not. **41**(1), 372–382 (2006)

12. Johns, M., Engelmann, B., Posegga, J.: XSSDS: server-side detection of cross-site scripting attacks. In: 2008 Annual Computer Security Applications Conference (ACSAC), pp. 335–344. IEEE (2008)
13. Fulton, N., Omar, C., Aldrich, J.: Statically typed string sanitation inside a Python. In: Proceedings of the 2014 International Workshop on Privacy & Security in Programming. ACM (2014)
14. Micheelsen, S., Thalmann, B.: A static analysis tool for detecting security vulnerabilities in python web applications (2016)
15. Giannopoulos, L., et al.: Pythia: identifying dangerous data-flows in Django-based applications. EuroSec@ EuroSys (2019)
16. Johns, M.: Towards practical prevention of code injection vulnerabilities on the programming language level (2007)
17. Johns, M., Beyerlein, C., Giesecke, R., Posegga, J.: Secure code generation for web applications. In: Massacci, F., Wallach, D., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 96–113. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11747-3_8
18. Johns, M.: Code-injection vulnerabilities in web applications — exemplified at cross-site scripting. IT Inf. Technol. Methoden Innov. Anwend. Inform. Inf. **53**(5), 256–260 (2011)