A Taint Mode for Python via a Library

[Anonymized for AppSec 2010]

Abstract. Vulnerabilities in web applications present threats to on-line systems. SQL injection and cross-site scripting attacks are among the most common threats found nowadays. These attacks are often result of improper or none input validation. To help discover such vulnerabilities, taint analyses have been developed in popular web scripting languages like Perl, Ruby, PHP, and Python. Such analysis are often implemented as an execution monitor, where the interpreter needs to be adapted to provide a taint mode. However, modifying interpreters might be a major task in its own right. In fact, it is very probably that new releases of interpreters require to be adapted to provide a taint mode. Differently from previous approaches, we show how to provide a taint analysis for Python via a library written entirely in Python, and thus avoiding modifications in the interpreter. The concepts of classes, decorators and dynamic dispatch makes our solution lightweight, easy to use, and particularly neat. With minimal or none effort, the library can be adapted to work with different Python interpreters.

1 Introduction

Over the past years, there has been a dramatic increase on the activities that can be performed on-line. Users can do almost everything using a web browser (e.g. watching videos, listening to music, banking, booking flights, planing trips, etc). Considering the size of Internet and its number of users, web applications are probably among the most used pieces of software nowadays. Despite its wide use, web applications suffer from vulnerabilities that permit attackers to steal confidential data, break integrity of systems, and affect availability of services. When development of web applications is done with little or no security in mind, the presence of security holes increases dramatically. Web-based vulnerabilities have already outplaced those of all other platforms [5] and there are no reasons to think that this tendency has changed [11].

According to OWASP [26], cross-site scripting (XSS) and SQL injection (SQLI) attacks are among the most common vulnerabilities on web applications. Although these attacks are classified differently, they are produced by the same reason: *user supplied data is sent to sensitive sinks without a proper sanitation*. For example, when a SQL query is constructed using an unsanitize string provided by a user, SQL injection attacks are likely to occur. To harden applications against these attacks, the implementations of some popular web script languages provide taint analyses in a form of execution monitors [2, 3]. In that manner, not only run interpreters code, but they also perform security checks. Taint analyses can also be provided through static analyses [14, 15]. Nevertheless, execution monitors usually produce less false alarms than traditional static techniques [23]. In other words, execution monitors are likely to be more precise than traditional static techniques. In particular, static techniques cannot deal with dynamic code evaluation without being too conservative. Most of the modern web scripting languages are capable to dynamically execute code. In this paper, we focus on dynamic techniques.

```
python email.py alice@domain.se ./reportJanuary.xls
python email.py devil@evil.com  '/etc/passwd'
python email.py devil@evil.com  '/etc/passwd ; rm -rf / '
```

Fig. 2. Different invocations for `email.py`

A taint analysis is an automatic approach to find vulnerabilities. Intuitively, a taint analysis restricts how tainted or untrustworthy data flow inside programs. Specifically, it constrains data to be untainted (trustworthy) or previously sanitized when reaching sensitive sinks. Perl was the first scripting language to provide a taint analysis as an special mode of the interpreter called *taint mode* [7]. Similar to Perl, some interpreters for Ruby [25], PHP [19], and recently Python [16] have been carefully modified to provide taint modes. Adapting interpreters to incorporate taint analyses present two major drawbacks that directly impact on the adoption of this technology. Firstly, incorporating a taint analysis into an interpreter might be a major task in its own right. Secondly, it is very probably that it is necessary to repeatedly adapt an interpreter at every new version or release of it.

Rather than modifying interpreters, we present how to provide a taint mode for Python via a library written entirely in Python. Python is spreading fast inside web development [1]. Besides its successful use, Python presents some programming languages abstractions that makes possible to provide a taint mode via a library. For example, Python decorators [18] are a non-invasive and simple manner to declare sources of tainted data, sensitive sinks, and sanitation functions. Python's object-oriented and dynamic typing mechanisms allows the execution of the taint analysis with almost no modifications in the source code.

```
import sys
import os

usermail = sys.argv[1]
file = sys.argv[2]

cmd = 'mail -s "Requested file" '
      + usermail + ' < ' + file
os.system(cmd)
```

Fig. 1. Code for `email.py`

The library provides a general method to enhance Python's built-in classes with tainted values. In general, taint analysis tends to only consider strings or characters [2, 19, 13, 16, 12, 24]. In contrast, our library can be easily adapted to consider different built-in classes and thus providing a taint analysis for a wider set of data types. By only considering tainted strings, the library provides a similar analysis than in [16], but without modifying the Python interpreter. To the best of our knowledge, a library for taint analysis has not been considered before.

1.1 A motivating example

We present an example to motivate the use of taint analysis in order to discover and repair vulnerabilities. The example considers an scenario of a web application where

users can send their remotely stored files by email. Figure 1 shows the simple module `email.py` that is responsible to perform such task. For simplicity, the code takes the user input from the command line (lines 4 and 5) rather than from the web server. Figure 2 shows some invocations to the module from the shell prompt. Line 1 shows a request from Alice to send her own file `reportJanuary.xls` to her email address `alice@domain.se`. In this case, Alice's input produces a behavior which matches the intention of the module. In contrast, line 2 and 3 shows how attackers can provide particular inputs to exploit unintended or unforeseen behaviors of `email.py`. Line 2 exploits the fact that `email.py` was written assuming that users only request their own files. Observe how `devil@evil.com` gets information regarding users accounts by receiving the file `/etc/passwd`. Line 3 goes an step further and injects the command `rm -rf /` after sending the email. These attacks demonstrate how, what was intended to be a simple email client, can become a web-based file browser or a terminal. To avoid this type of vulnerabilities, applications need to rigorously check for malicious data from any input coming from users or any other untrustworthy source. Taint analysis helps to detect when data is not sanitize before it is used on security critical operations. In Section 2.2, we show how to harden `email.py` in order to reject the type of vulnerabilities shown in Figure 1.

The paper is organized as follows. Section 2 outlines the library API. Section 3 describes the most important implementation details of our approach. Section 4 covers related work. Section 5 provides some concluding remarks.

2 A library for taint analysis

On most situations, taint analyses propagate taint information on assignments. Intuitively, when the right-hand side of an assignment uses a tainted value, the variable appearing on the left-hand side becomes tainted. Taint analyses can be seen as information-flow tracking mechanisms for integrity [22]. In fact, taint analyses are simply mechanisms to track explicit flows, i.e. direct flows of information from one variable to another. Taint analyses tend to ignore implicit flows [10], i.e. flows through the control-flow constructs of the language. Figure 3 presents an implicit flow. Variables `t` and `u` are tainted and untainted, respectively. Observe that variable `u` is untainted after the execution of the branch since an untainted value (`'a'` or `''`) is assigned to it. Yet, the value of the tainted variable `t` is copied into the untainted variable `u` when `t == 'a'`. It is not difficult to imagine programs that circumvent the taint analysis by copying the content of tainted strings into untainted ones by using implicit flows[21].

```
if t == 'a': u = 'a'
else: u = ''
```

Fig. 3. An implicit flow

In scenarios where attackers has full control over the code (e.g. when the code is potentially malicious), implicit flows present an effective way to circumvent the taint analysis. In this case, the attackers' goal is to craft the code and input data in order to circumvent security mechanisms. There is a large body of literature on the area of language-based security regarding how to track implicit flows [22].

There exists scenarios where the code is nonmalicious, i.e. written without malice. Despite the good intentions and experience of programmers, the code might still contain

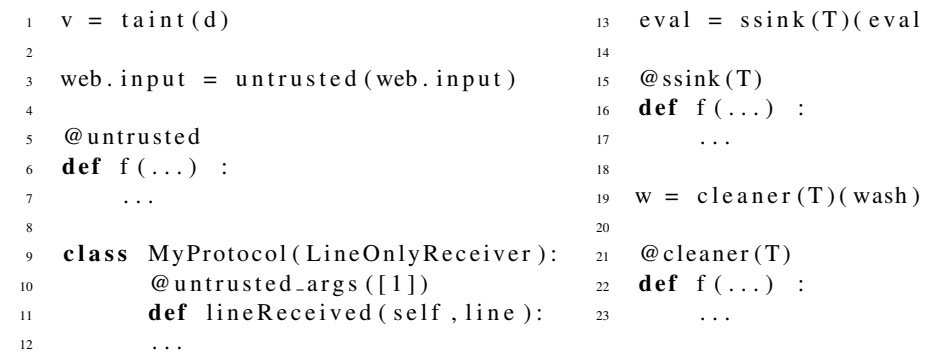

Fig. 4. API for taint analysis

vulnerabilities as the ones described in Section 1.1. The attackers' goal consists on craft input data in order to exploit vulnerabilities and/or corrupt data. In this scenario, taint analysis certainly helps to discover vulnerabilities. How dangerous are implicit flows in nonmalicious code? We argue that they are frequently harmless [21]. The reason for that relies on that nonmalicious programmers need to write a more involved, and rather unnatural, code in order to, for instance, copy tainted strings into untainted ones. In contrast, to produce explicit flows, programmers simply need to forget a call to some sanitization function. For the rest of the paper, we consider scenarios where the analyzed code is non-malicious.

2.1 Using the library

The library is just a series of functions to mark what are the sources of untrustworthy data, sensitive sinks, and sanitation functions. Figure 4 illustrates how the API works. Symbol `...` is a place holder for code that is not relevant to explain the purpose of the API. We assume that `v` is a variable, `d` is an string or integer, and `f` is a user-defined function. Symbol `T` represents a tag. By default, tags can take values `XSS`, `SQLI`, `OSI` (Operating System Injection), and `II` (Interpreter Injection). These values are used to indicate specific vulnerabilities that could be exploited by tainted data. For instance, tainted data associated with tag `SQLI` is likely to exploit SQL injection vulnerabilities. Function `taint` is used to taint values. For example, line 1 taints variable `d`. The call to `untrusted(web.input)` establishes that the results produced by `web.input` are tainted. Line 5 shows how `untrusted` can be used to mark the values returned by function `f` as untrustworthy. Observe the use of the decorator syntax (`@untrusted`). Function `untrusted_args` is used to indicate which functions' arguments must be tainted. This primitive is particularly useful when programming frameworks require to redefine some methods in order to get information from external sources into the application. As an example, Twisted[4], a framework to develop network applications, calls method `lineReceived` from the class `LineOnlyReceiver` every time that an string is received from the network. Line 9–12 extend the class `LineOnlyReceiver`

and implement the method `lineReceived`. Line 10 taints the data that Twisted takes from the network. Functions `taint`, `untrusted`, and `untrusted_args` associate all the tags to the tainted values. After all, untrustworthy data might exploit any kind of vulnerability. Line 13 marks `eval` as a sensitive sink. If eval receives a tainted data with the tag `T`, a possible vulnerability T is reported. Line 15 shows how to use `ssink` with the decorator syntax. Line 19 shows how `cleaner` establishes that function `wash` sanitizes data with tag `T`. As a result of that, function `w` removes tag `T` from tainted values. Line 21 shows the use of `cleaner` with the decorator syntax. Sensitive sinks and sanitization functions can be associated with more than one kind of vulnerabilities by just nesting decorators, i.e. `ssink(OSI)(ssink(II)(critical_operation))`.

2.2 Hardening `email.py`

We revise the example in Section 1.1. Figure 5 shows the secure version of the code given in Figure 1. Line 3 imports the library API. Line 4 imports some sanitization functions. Line 6 marks command `os.system` (capable to run arbitrary shell instructions) as a sensitive sink to `OSI` attacks. Tainted values reaching that sink must not contain the tag `OSI`. Lines 7 and 8 establish that functions `s_usermail` and `s_file` sanitize data in order to avoid `OSI` attacks. Lines 10 and 11 mark user input as untrustworthy. When executing the program, the taint analysis raises an alarm on line 16. The reason for that is that variable `cmd` is tainted with the tag `OSI`. Indeed, `cmd` is formed from the untrustworthy values `usermail` and `file`. If we uncomment the lines where sanitization takes place (lines 12 and 13), the program runs normally, i.e. no alarms are reported. Observe that the main part of the code (lines 14–16) are the same than in Figure 1.

```
import sys                                              1
import os                                               2
from taintmode import *                                 3
from sanitize  import *                                 4
                                                        5
os.system = ssink(OSI)(os.system)                       6
s_usermail = cleaner(OSI)(s_usermail)                   7
s_file = cleaner(OSI)(s_file)                           8
                                                        9
usermail = taint(sys.argv[1])                           10
file = taint(sys.argv[2])                               11
#usermail = s_usermail(usermail)                        12
#file = s_file(file)                                    13
cmd = 'mail -s "Requested file" '                       14
      + usermail + ' < ' + file                         15
os.system(cmd)                                          16
```

Fig. 5. Secure version of module `email.py`

3 Implementation

In this section we present the details of our implementation. Due to lack of space, we show the most interesting parts. The full implementation of the library is publicly available at [6].

One of the core part of the library deals with how to keep track of taint information for built-in classes. The library defines subclasses of built-in classes in order to indicate if values are tainted or not. An object of these subclasses posses an attribute to indicate

```
1  def taint_class(klass, methods):
2      class tklass(klass):
3        def __new__(cls, *args, **kwargs):
4            self = super(tklass, cls).__new__(cls, *args, **kwargs)
5            self.taints = set()
6            return self
7      d = klass.__dict__
8      for name, attr in [(m, d[m]) for m in methods]:
9          if inspect.ismethod(attr) or
10            inspect.ismethoddescriptor(attr):
11               setattr(tklass, name, propagate_method(attr))
12     if '__add__' in methods and '__radd__' not in methods:
13         setattr(tklass, '__radd__',
14                 lambda self, other: tklass.__add__(tklass(other),
15                 self))
16     return tklass
```

Fig. 6. Function to generate taint-aware classes

a set of tags associated to it. Objects are considered untainted when the set of tags is empty. We refer to these subclasses as *taint-aware classes*. In addition, the methods inherited from the built-in classes are redefined in order to propagate taint information. More specifically, methods that belong to taint-aware classes return objects with the union of tags found in the arguments and in the object calling the method. In Python, the dynamic dispatch mechanism guarantees that, for instance, the concatenations of untainted and tainted strings is carried out with calls to methods of taint-aware classes, which properly propagates taint information.

3.1 Generating taint-aware classes

Figure 6 presents a function to generate taint-aware classes. The function takes a built-in class (`klass`) and a list of its methods (`methods`) where taint propagation must be performed. Line 2 defines the name of the taint-aware class `tklass`. Objects of `tklass` are associated to the empty set of tags when created (lines 3–6). Attribute `taints` is introduced to indicate the tags related to tainted values. Using Python's introspection features, variable `d` contains, among other things, the list of methods for the built-in class (line 7). For each method in the built-in class and

```
def propagate_method(method):                    1
    def inner(self, *args, **kwargs):            2
        r = method(self, *args, **kwargs)        3
        t = set()                                4
        for a in args:                           5
            collect_tags(a, t)                   6
        for v in kwargs.values():                7
            collect_tags(v, t)                   8
        t.update(self.taints)                    9
        return taint_aware(r,t)                  10
    return inner                                 11
```

Fig. 7. Propagation of taint information

in `methods` (lines 8–10), the code adds to `tklass` a method that has the same name and computes the same results but also propagates taint information (line 11). Function `propagate_method` is explained below. Line 12–15 set method `__radd__` to taint-aware classes when built-in classes do not include that method but `__add__`. Method `__radd__` is called to implement the binary operations with reflected (swapped) operands[1]. For instance, to evaluate the expression `x+y`, where `x` is a built-in string and `y` is a taint-aware string, Python calls `__radd__` from `y` and thus executing `y.__radd__(x)`. In that manner, the taint information of `y` is propagated to the expression. Otherwise, the method `x.__add__(y)` is called instead, which results in an untainted string. Finally, the taint-aware class is returned (line 16).

The implementation of `propagate_method` is shown in Figure 7. The function takes a method and returns another method that computes the same results but propagates taint information. Line 3 calls the method received as argument and stores the results in `r`. Lines 4–9 collects the tags from the current object and the method's arguments into `t`. Variable `r` might refer to an object of a built-in class and therefore not include the attribute `taints`. For that reason, function `taint_aware` is designed to transform objects from built-in classes into taint-aware ones. For example, if `r` refers to a list of objects of the class `str`, function `taint_aware` returns a list of objects of the taint-aware class derived from `str`. Function `taint_aware` is essentially implemented as a structural mapping on list, tuples, sets, and dictionaries. The library does not taint built-in containers, but rather their elements. This is a design decision based on the assumption that nonmalicious code does not exploit containers to circumvent the taint analysis (e.g. by encoding the value of tainted integers into the length of lists). Otherwise, the implementation of the library can be easily adapted. Line 11 returns the taint-aware version of `r` with the tags collected in `t`.

```
STR = taint_class(str, str_methods)
INT = taint_class(int, int_methods)
```

Fig. 8. Taint-aware classes for strings and integers

To illustrate how to use function `taint_class`, Figure 8 produces taint-aware classes for strings and integers, where `str_methods` and `int_methods` are lists of methods for the classes `str` and `int`, respectively. Observe how the code presented in Figures 6 and 7 is general enough to be applied to several built-in classes.

3.2 Decorators

Except for `taint`, the rest of the API is implemented as decorators. In our library, decorators are high order functions [8], i.e. functions that take functions as arguments and return functions. Figure 9 shows the code for `untrusted`. Function f, given as an argument, is the function that returns untrustworthy results (line

```
1  def untrusted(f):
2      def inner(*args, **kwargs):
3          r = f(*args, **kwargs)
4          return taint_aware(r, TAGS)
5      return inner
```

Fig. 9. Code for `untrusted`

[1] The built-in class for strings implements all the reflected versions of its operators but `__add__`.

1). Intuitively, function `untrusted` returns a function (`inner`) that calls function `f` (line 3) and taints the values returned by it (line 4). Symbol `TAGS` is the set of all the tags used by the library. Readers should refer to [6] for further details about the implementation of the API.

3.3 Taint-aware functions

Several dynamic taint analysis [2, 19, 15, 16, 12, 24] do not propagate taint information when results different from strings are computed from tainted values. (e.g. the length of a tainted string is usually an untainted integer). This design decision might affect the abilities of taint analysis to detect vulnerabilities. For instance, taint analysis might miss dangerous patterns when programs encode strings as lists of numbers. A common workaround to this problem is to mark functions that perform encodings of strings as sensitive sinks. In that manner, sanitization must occur before strings are represented in another format. Nevertheless, this approach is unsatisfactory: the intrinsic meaning of sensitive sinks may be lost. Sensitive sinks are security critical operations rather than functions that perform encodings of strings. Our library provides means to start breaching this gap.

```
1  def propagate_func(original):
2      def inner(*args, **kwargs):
3          t = set()
4          for a in args:
5              collect_tags(a, t)
6          for v in kwargs.values():
7              collect_tags(v, t)
8          r = original(*args,**kwargs)
9          if t == set([]):
10             return r
11         return taint_aware(r, t)
12     return inner
```

Fig. 10. Propagation of taint information among possibly different taint-aware objects

Figure 10 presents a general function that allows to define operations that return tainted values when their arguments involve taint-aware objects. As a result, it is possible to define functions that, for instance, take tainted strings and return tainted integers. We classify this kind of functions as *taint-aware*.

```
len = propagate_func(len)
ord = propagate_func(ord)
chr = propagate_func(chr)
```

Fig. 11. Taint-aware functions for strings and integers

Similar to the code shown in Figure 7, `propagate_func` is a high order function. It takes function `f` and returns another function (`inner`) able to propagate taint information from the arguments to the results. Lines 3–7 collect tags from the arguments. If the set of collected tags is empty, there are no tainted values involved and therefore no taint propagation is performed (lines 9–10). Otherwise, a taint-aware version of the results is returned with the tags collected in the arguments (line 11).

To illustrate the use of `propagate_func`, Figure 11 shows some taint-aware functions for strings and integers. We redefine the standard functions to compute lengths of lists (`len`), the ASCII code of a character (`chr`), and its inverse (`ord`). As a result, `len(taint('string'))` returns the tainted integer 6. It is up to the users of the library to decide which functions must be taint-aware depending on the scenario. The

library only provides redefinition of standard functions like the ones shown in Figure 11.

3.4 Scope of the library

In Figure 6, the method to automatically produce taint-aware classes does not work with booleans. The reason for that is that class `bool` cannot be subclassed in Python [2]. Consequently, our library cannot handle tainted boolean values. We argue that this shortcoming does not restrict the usability of the library for two reasons. Firstly, different from previous approaches [2, 19, 15, 16, 12, 24], the library can provide taint analysis for several built-in types rather than just strings. Secondly, we consider that booleans are typically used on guards. Since the library already ignores implicit flows, the possibilities to find vulnerabilities are not drastically reduced by disregarding taint information on booleans.

When generating the taint-aware class `STR` (Figure 8), we found some problems when dealing with some methods from the class `str`. Python interpreter raises exceptions when methods `__nonzero__`, `__reduce__`, and `__reduce_ex__` are redefined. Moreover, when methods `__new__`, `__init__`, `__getattribute__`, and `__repr__` are redefined by function `taint_class`, an infinite recursion is produced when calling any of them. As for `STR`, the generation of the taint-aware class `INT` exposes the same behavior, i.e. the methods mentioned before produce the same problems. We argue that this restriction does not drastically impact on the capabilities to detect vulnerabilities. Methods `__new__` is called when creating objects. In Figure 6, taint-aware classes define this method on line 3. Method `__init__` is called when initializing objects. Python invokes this method after an object is created and programs do not usually called it explicitly. Method `__getattribute__` is used to access any attribute on a class. This method is automatically inherited from `klass` and it works as expected for taint-aware classes. Method `__nonzero__` is called when objects need to be converted into a boolean value. As mentioned before, the analysis ignores taint information of data that is typically used on guards. Method `__repr__` pretty prints objects on the screen. In principle, developers should be careful to not use calls to `__repr__` in order to convert tainted objects into untainted ones. However, this method is typically used for debugging [3]. Methods `__reduce__` and `__reduce_ex__` are used by Pickle [4] to serialize strings. Given these facts, the argument `method` in function `taint_class` establishes the methods to be redefined on taint-aware classes (Figure 6). This argument is also useful when the built-in classes might vary among different Python interpreters. It is future work to automatically determine the lists of methods to be redefined for different built-in classes and different versions of Python.

It is up to the users of the library to decide which built-in classes and functions must be taint-aware. This attitude comes from the need of being flexible and not affecting performance unless it is necessary. Why users interested on taint analysis for strings should accept run-time overheads due to tainted integers?

[2] `http://docs.python.org/library/functions.html#bool`
[3] `http://docs.python.org/reference/datamodel.html`
[4] An special Python module

As a future work, we will explore if it is possible to automatically define taint-aware functions based on the built-in functions (found in the interpreter) and taint-aware classes in order to increase the number of taint-aware functions provided by the library. At the moment, the library provides taint-aware classes for strings, integers, floats, and unicode as well as some taint-aware functions (e.g. `len`, `chr`, and `ord`).

4 Related Work

Perl [2] was the first scripting language to include taint analysis as a native feature of the interpreter. Perl taint mode marks strings originated from outside a program as tainted (i.e. inputs from users, environment variables, and files). Sanitization is done by using regular expressions. Writing to files, executing shell commands, and sending information over the network are considered sensitive sinks. Differently, our library gives freedom to developers to classify the sources of tainted data, sanitization functions, and sensitive sinks. Similar to Perl, Ruby [25] provides support for taint analysis. Ruby's taint mode, however, performs analysis at the level of objects rather than only strings. Both, Perl and Ruby, utilize dynamic techniques for their analyses.

Several taint analysis have been developed for the popular scripting language PHP. Aiming to avoid any user intervention, authors in [14] combine static and dynamic techniques to automatically repair vulnerabilities in PHP code. They propose to use static analysis to insert some predetermined sanitization functions when tainted values reach sensitive sinks. Observe that the semantic of programs is changed by inserting calls to sanitization functions. In contrast, our approach only reports vulnerabilities and it is up to developers to decide where, and how, sanitization procedures must be called. In [19], Nguyen-Toung et al. adapt the PHP interpreter to provide a dynamic taint analysis at the level of characters, which the authors call *precise tainting*. They argue that precise tainting gains precision over traditional taint analyses for strings. Authors need to manually exploit, when feasible, semantics definitions of functions in order to accurately keep track of tainted characters. Our approach, on the other hand, uses the same mechanism to handle tainted values independently of the nature of a given function. Consequently, we are able to automatically extend our analysis to different set of data types but without being as precise as Nguyen-Toung et al.' work. It is worth seeing studies indicating how much precision (i.e. less false alarms) it is obtained with *precise tainting* in practice. Similarly to Nguyen-Toung et al.'s work, Futoransky [12] et al. provides a precise dynamic taint analysis for PHP. Jovanovic et al. [15] proposes to combine a traditional data flow and alias analysis to increase the precision of their static analysis. They observe a 50% rate of false alarms (i.e. one false alarm for each vulnerability).

A taint analysis for Java [13] instruments the class `java.lang.String` as well as classes that present untrustworthy sources and sensitive sinks. The instrumentation of `java.lang.String` is done offline, while other classes are instrumented online. The authors mention that a custom class loader in the JVM is needed in order to perform online instrumentation.

A series of work [17, 9, 20] propose to provide information-flow security via a library in Haskell. These libraries handle explicit and implicit flows and programmers

need to write programs with an special-purpose API. Similar to other taint analyses, our library does not contemplate implicit flows and programs do not need to be written with an special-purpose API.

Among the closest related work, we can mention [16] and [24]. In [16], authors modify the Python interpreter to provide a dynamic taint analysis. More specifically, the representation of the class `str` is extended to include a boolean flag that marks if a string is tainted. We provide a similar analysis but without modifying the interpreter. The work by Seo and Lam [24], called InvisiType, aims to enforce safety checks without modifying the analyzed code. Similar to our assumptions, their approach is designed to work with nonmalicious code. InvisiType is more general than our approach. In fact, authors shows how InvisiType can provide taint analysis and access control checks for Python programs. However, InvisiType relies on several modifications in the Python interpreter in order to perform the security checks at the right places. For example, when native methods are called, the run-time environment firstly calls the special purpose method `__nativecall__`. As a manner to specifying policies, the approach provides the class `InvisiType` that defines special purposes methods to get support from the run-time system (e.g. `__nativecall__` is one of those methods). Subclasses of this class represent security policies. The approach relies on multiple inheritance to extend existing classes with security checks. To include or remove security checks from objects, programs needs to explicitly call functions *promote* and *demote*. Being less invasive, our library uses decorators instead of explicit function calls to taint and untaint data. Our approach does not require multiple inheritance.

5 Conclusions

We propose a taint mode for Python via a library entirely written in Python. We show that no modifications in the interpreter are needed. Different from traditional taint analysis, our library is able to keep track of tainted values for several built-in classes. Additionally, the library provide means to define functions that propagate taint information (e.g. the length of a tainted string produces a tainted integer). The library consists on around 300 LOC. To apply taint analysis in programs, it is only needed to indicate the sources of untrustworthy data, sensitive sinks, and sanitization functions. The library uses decorators as a noninvasive approach to mark source code. Objects classes and dynamic dispatch allow the analysis to execute with almost no modifications in the code. As a future work, we plan to use the library to harden frameworks for web development and evaluate the capabilities of our library to detect vulnerabilities in popular web applications.

References

[1] List of Python software. `http://en.wikipedia.org/wiki/List_of_Python_software`.

[2] The Perl programming language. `http://www.perl.org/`.

[3] The Ruby programming language. `http://www.ruby-lang.org/en/`.

[4] The Twisted programming framework. `http://twistedmatrix.com/trac/`.

[5] M. Andrews. Guest Editor's Introduction: The State of Web Security. *IEEE Security and Privacy*, 4(4):14–15, 2006.

[6] Anonymized. Taintlib. Software release. `http://taintmode.firebirds.com.ar/taintmode.txt`, Feb. 2010.

[7] S. Bekman and E. Cholet. *Practical mod_perl*. O'Reilly and Associates, 2003.

[8] R. Bird and P. Wadler. *An introduction to functional programming*. Prentice Hall International (UK) Ltd., 1988.

[9] T. chung Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. *Computer Security Foundations Symposium, IEEE*, 0:187–202, 2007.

[10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[11] Federal Aviation Administration (US). Review of Web Applications Security and Intrusion Detection in Air Traffic Control Systems. `http://www.oig.dot.gov/sites/dot/files/pdfdocs/ATC_Web_Report.pdf`, June 2009. Note: thousands of vulnerabilities were discovered.

[12] A. Futoransky, E. Gutesman, and A. Waissbein. A dynamic technique for enhancing the security and privacy of web applications. In *Black Hat USA Briefings*, Aug. 2007.

[13] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, 2005.

[14] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM, 2004.

[15] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *2006 IEEE Symposium on Security and Privacy*, pages 258–263. IEEE Computer Society, 2006.

[16] D. Kozlov and A. Petukhov. Implementation of Tainted Mode approach to finding security vulnerabilities for Python technology. In *Proc. of Young Researchers' Colloquium on Software Engineering (SYRCoSE)*, June 2007.

[17] P. Li and S. Zdancewic. Encoding information flow in haskell. *Computer Security Foundations Workshop, IEEE*, 0:16, 2006.

[18] M. Lutz and D. Ascher. *Learning Python*. O'Reilly & Associates, Inc., 1999.

[19] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[20] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 13–24. ACM, 2008.

[21] A. Russo, A. Sabelfeld, and K. Li. Implicit flows in malicious and nonmalicious code. *2009 Marktoberdorf Summer School (IOS Press)*, 2009.

[22] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[23] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.

[24] J. Seo and M. S. Lam. InvisiType: Object-Oriented Security Policies. In *17th Annual Network and Distributed System Security Symposium*. Internet Society (ISOC), Feb. 2010.

[25] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby. The Pragmatic Programmer's Guide*. Pragmatic Programmers, 2004.

[26] A. van der Stock, J. Williams, and D. Wichers. OWASP Top 10 2007. `http://www.owasp.org/index.php/Top\_10\_2007`, 2007.