pytaint

Intern: Marcin Fatyga <fatygamarcin@gmail.com>
Host: Felix Gröbert <groebert@google.com>

public version

Agenda

1. Taint tracking overview.
2. Pytaint design.
   a. taint representation
   b. enabling taint tracking in apps
   c. rough edges in design
3. Pytaint implementation overview.
4. Summary

Taint tracking - overview

- attaching metadata to variables

Taint tracking - overview

- attaching metadata to variables
- can be used to prevent various types of attacks

Taint tracking - overview

- attaching metadata to variables
- can be used to prevent various types of attacks
- old idea:
  - ...Ruby has had it for 10+ years
  - ...Perl has had it since 1989

Taint tracking - overview

- attaching metadata to variables
- can be used to prevent various types of attacks
- old idea:
  - ...Ruby has had it for 10+ years
  - ...Perl has had it since 1989
  - ...libraries for other languages, including Python

Taint tracking - overview

- attaching metadata to variables
- can be used to prevent various types of attacks
- old idea:
  - ...Ruby has had it for 10+ years
  - ...Perl has had it since 1989
  - ...libraries for other languages, including Python

But noone uses it. Why?

Problems

- inflexible - taint is binary
- programmer have to remember about checks, tainting, cleaning...
- performance issues (in existing Python implementation)

How to make usable taint tracking?

- flexible taint
- global configurations for applications
- changes to Python builtins

How to make usable taint tracking?

- flexible taint
  - ...based heavily on ideas presented by Meder Kydyraliev in his Gravizapa project
- global configurations for applications
- changes to Python builtins

Taint semantics overview

- all operations on tainted objects result in tainted objects
- a tainted string can gain merit
- merit is a “security contract”
- each merit indicates that the object is safe for some specific operation
- merits also propagate in string operations

String extension example

```
s = “foo”
s.istainted() # False
```

variable	contents	safe for:
s	“foo”	everything

String extension example

```
s = “foo”
s.istainted() # False

s = s.taint()
s.istainted() # True
```

variable	contents	safe for:
s	“foo”	nothing

String extension example

```
s = “foo”
s.istainted() # False

s = s.taint()
s.istainted() # True

s = s.upper()
```

variable	contents	safe for:
s	“FOO”	nothing

String extension example

```
s = “foo”
s.istainted() # False

s = s.taint()
s.istainted() # True

s = s.upper()
p = “bar”
r = s + p
```

variable	contents	safe for:
s	“FOO”	nothing
p	“bar”	everything
r	“FOO bar”	nothing

String extension example

```
s = “foo”
s.istainted() # False

s = s.taint()
s.istainted() # True

s = s.upper()
p = “bar”
r = s + p
s = s._cleanfor(SQLiMerit)
```

variable	contents	safe for:
s	“FOO”	SQL queries
p	“bar”	everything
r	“FOO bar”	nothing

String extension example

```
s = “foo”
s.istainted() # False

s = s.taint()
s.istainted() # True

s = s.upper()
p = “bar”
r = s + p
s = s._cleanfor(SQLiMerit)
s = s._cleanfor(ShellMerit)
```

variable	contents	safe for:
s	“FOO”	SQL queries, running in shell
p	“bar”	everything
r	“FOO bar”	nothing

String extension example

```
s = “foo”
s.istainted() # False

s = s.taint()
s.istainted() # True

s = s.upper()
p = “bar”
r = s + p
s = s._cleanfor(SQLiMerit)
s = s._cleanfor(ShellMerit)

p.isclean(SQLiMerit) # True
r.isclean(SQLiMerit) # False
s.isclean(SQLiMerit) # True
```

variable	contents	safe for:
s	“FOO”	SQL queries, running in shell
p	“bar”	everything
r	“FOO bar”	nothing

How to make taint tracking usable?

- flexible taint
- global configurations for applications
  - (again, some ideas borrowed from Gravizapa)
- changes to Python builtins

Patching applications

```
import pipes
import os



s = raw_input()

# pipes.quote returns shell escaped version of s
s = pipes.quote(s)

c = 'whois ' + s

os.system(c)
```

Patching applications

```
import pipes
import os
from taint import ShellMerit, TaintException



s = raw_input()
s = s.taint()
# pipes.quote returns shell escaped version of s
s = pipes.quote(s)
s = s._cleanfor(ShellMerit)
c = 'whois ' + s
if c.isclean(ShellMerit):
    os.system(c)
else:
    raise TaintException
```

Patching applications

```
import pipes
from taint import taint, ShellMerit, TaintException

s = raw_input()
s = s.taint()
# pipes.quote returns shell escaped version of s
s = pipes.quote(s)
s = s._cleanfor(ShellMerit)
c = 'whois ' + s
if c.isclean(ShellMerit):
    os.system(c)
else:
    raise TaintException
```

Easy to forget about

LOC increased by 100%!

This looks ugly, but probably will be a common idiom

Patching applications

```
import pipes
from taint import taint, ShellMerit, TaintException

s = raw_input()
s = s.taint()
# pipes.quote returns shell escaped version of s
s = pipes.quote(s)
s = s._cleanfor(ShellMerit)
c = 'whois ' + s
if c.isclean(ShellMerit):
    os.system(c)
else:
    raise TaintException
```

Easy to forget about

Not cool.

LOC increased by 100%!

This looks ugly, but probably will be a common idiom

Patching applications

```
import pipes
import os
from taint import ShellMerit, TaintException



s = raw_input()
s = s.taint()
# pipes.quote returns shell escaped version of s
s = pipes.quote(s)
s = s._cleanfor(ShellMerit)
c = 'whois ' + s
if c.isclean(ShellMerit):
    os.system(c)
else:
    raise TaintException
```

Google

Taint tracking - high level concepts

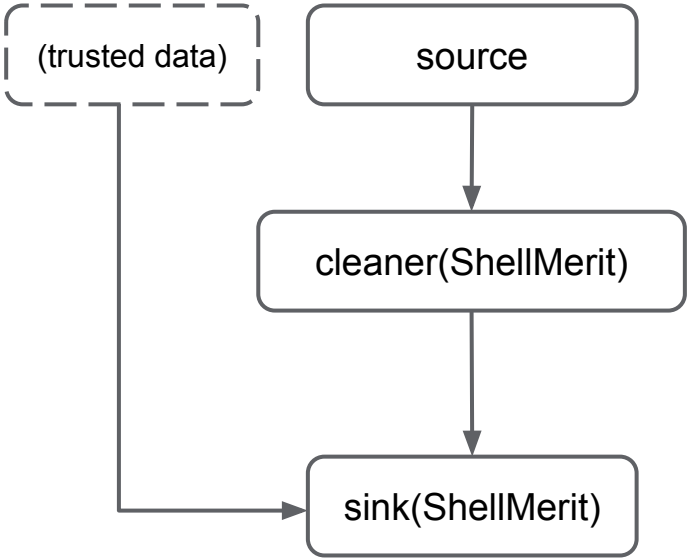
```
import pipes
import os
from taint import ShellMerit, TaintException


s = raw_input()
s = s.taint()
# pipes.quote returns shell escaped version of s
s = pipes.quote(s)
s = s._cleanfor(ShellMerit)
c = 'whois ' + s
if c.isclean(ShellMerit):
    os.system(c)
else:
    raise TaintException
```

A tainted data **source**.

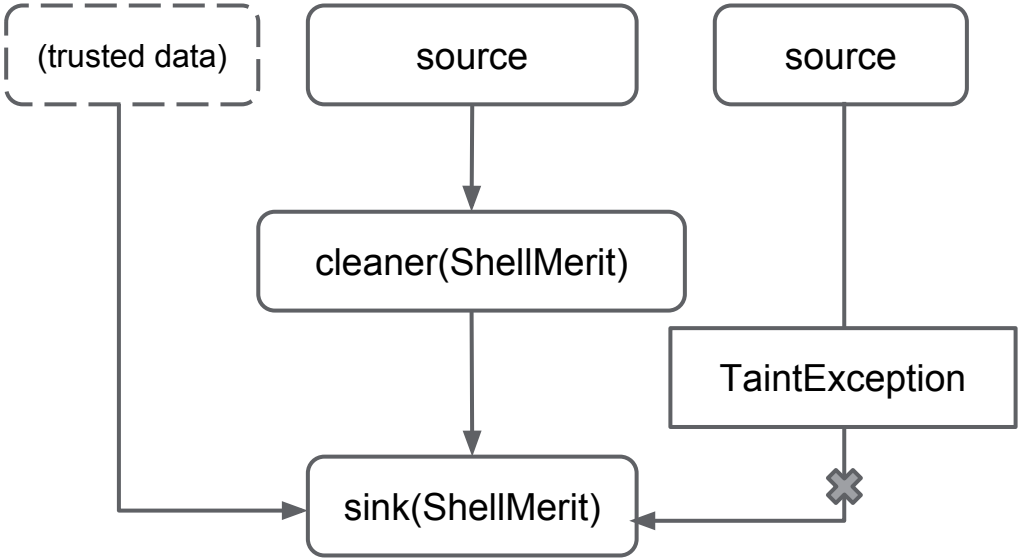
ShellMerit **cleaner**.

A **sink** sensitive to ShellMerits.

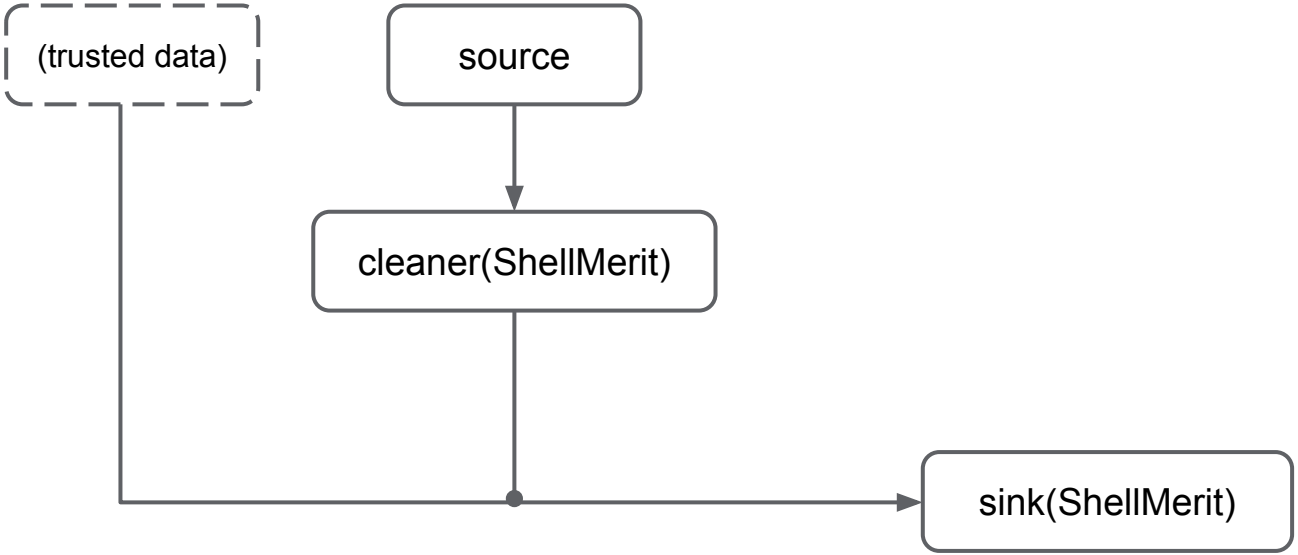
Dataflow - what it should look like



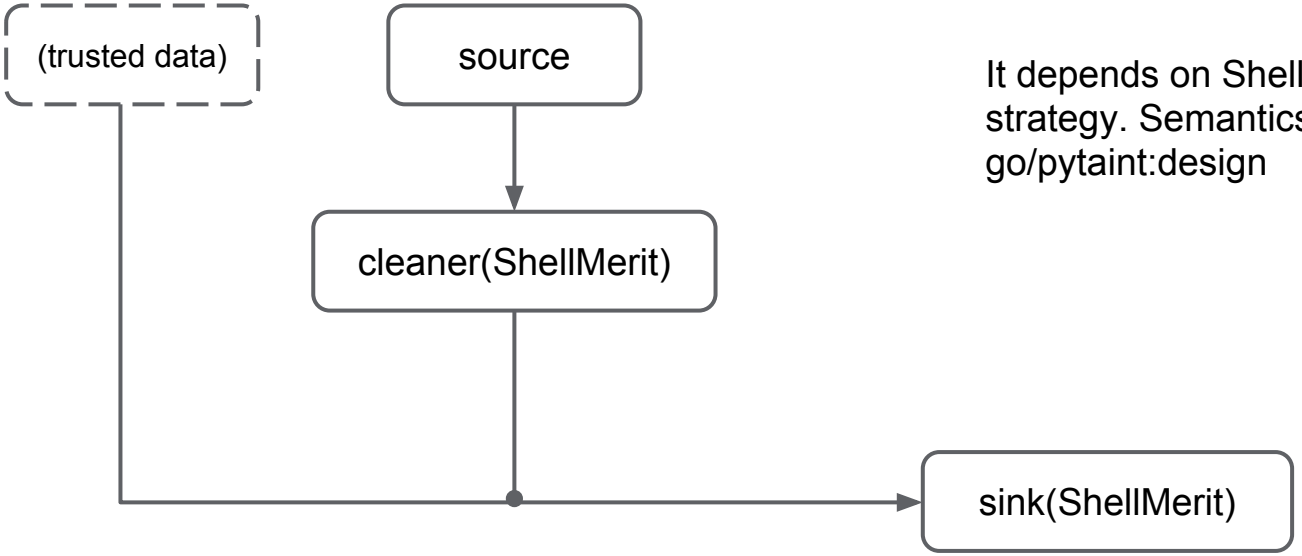
Dataflow - taint violation



Dataflow - what should happen?



Dataflow - what should happen?



It depends on ShellMerit's propagation strategy. Semantics are defined in go/pytaint:design

Taint tracking - high level concepts

```
import pipes
import os
from taint import ShellMerit, TaintException
```

```
s = raw_input()
```

A tainted data **source**.

```
# pipes.quote returns shell escaped version of s
s = pipes.quote(s)
```

ShellMerit **cleaner**.

```
c = 'whois ' + s

os.system(c)
```

A **sink** sensitive to ShellMerits.

Config files

```
{
    "sources": [
        "raw_input"
    ],
    "cleaners": [
        {
            "merit": "ShellMerit"
        },
        "pipes.quote"
    ],
    "sinks": [
        {
            "merit": ShellMerit
        },
        "os.system"
    ]
}
```

Intended usage

```
import pipes
import os



s = raw_input()

# pipes.quote returns shell escaped version of s
s = pipes.quote(s)

c = 'whois ' + s

os.system(c)
```

Intended usage

```
import pipes
import os
import taint

taint.enable("/tmp/config_shell.json")

s = raw_input()

# pipes.quote returns shell escaped version of s
s = pipes.quote(s)

c = 'whois ' + s

os.system(c)
```

Rough edges

- C extensions could break it
- some non intuitive design decisions

Rough edges - hashing

```
> 'abc'.__hash__() == 'abc'.taint().__hash__()
```

Rough edges - hashing

```
> 'abc'.__hash__() == 'abc'.taint().__hash__()
True
```

Rationale: don't break dictionaries.

Rough edges - equality

```
> ‘abc’.taint() == ‘abc’
```

Rough edges - equality

```
> 'abc'.taint() == 'abc'
True
```

Rationale: don't break conditionals

Rough edges - serialisation

```
> s = 'abc'.taint()
> json.loads(json.dumps(s)) == s
```

Rough edges - serialisation

```
> s = 'abc'.taint()
> json.loads(json.dumps(s)) == s
False
```

Rationale: no obvious solution for that.

Rough edges - control flow

```
if user == “admin”:
    s = sensitive_data
else:
    s = “forbidden”
```

Taint will not propagate from user to s.

How to make taint tracking usable?

- flexible taint
- global configurations for applications
- changes to Python builtins

Implementation

Two parts:

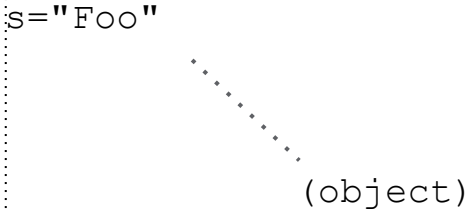
- patch to CPython
- taint.py module for standard library

CPython patching

- extending underlying C structures
  - propagation for all string and unicode methods
- rather time consuming
- 5% performance overhead

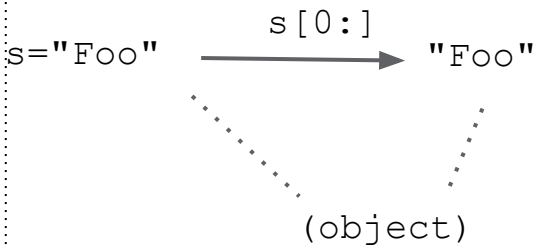
CPython patching

- changes in strings behaviour
- bigger memory consumption



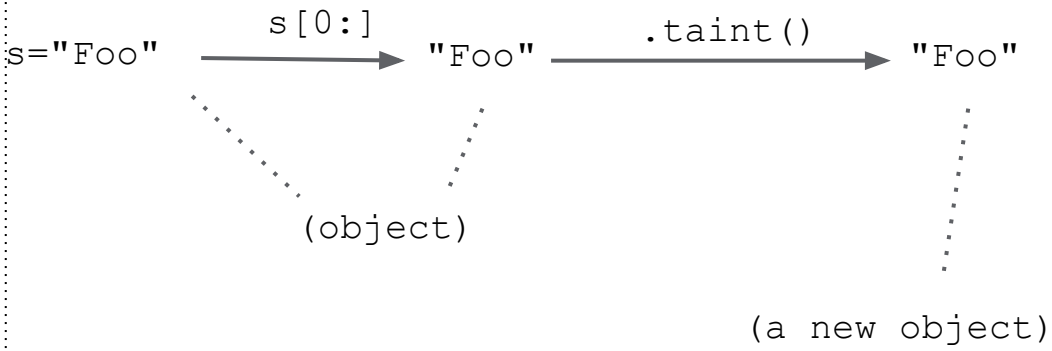
CPython patching

- changes in strings behaviour
- bigger memory consumption



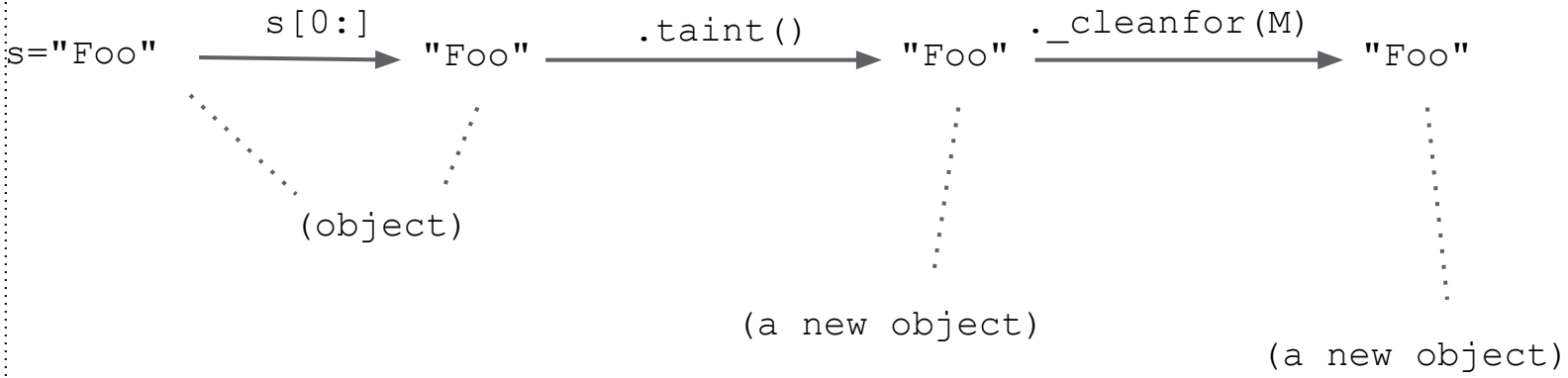
CPython patching

- changes in strings behaviour
- bigger memory consumption



CPython patching

- changes in strings behaviour
- bigger memory consumption



taint.py contents

Useful tools:

- taint patcher (taint.enable)
- decorators for sinks/sources/cleaners
- collection of Merits
- Propagator class proxy
  - adds taint propagation to “tricky” objects (like regular expressions)

Summary

- Goal: Python with taint tracking
  - Design
  - Implementation
    - reasonably fast
    - hopefully easy to use