



Taint Mode en Python

Juan José Conti

jjconti@gmail.com | FRSF UTN

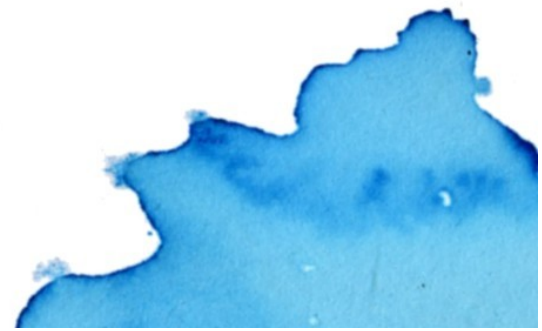
OWASP TOP 10

- * A1: Injection
- * A2: Cross-Site Scripting (XSS)
- * A3: Broken Authentication and Session Management
- * A4: Insecure Direct Object References
- * A5: Cross-Site Request Forgery (CSRF)
- * A6: Security Misconfiguration
- * A7: Insecure Cryptographic Storage
- * A8: Failure to Restrict URL Access
- * A9: Insufficient Transport Layer Protection
- * A10: Unvalidated Redirects and Forwards

El objetivo del atacante es: enviar entradas modificadas para tomar cierto control sobre operaciones del sistema

Consecuencias de no validar apropiadamente las entradas


- **Robo de identidad**
 - ID de sesiones alojados en cookies
- **Compromiso de información** confidencial
 - Acceso a información almacenada en una base de datos detrás de una aplicación web
- **Ataques de Denegación de Servicio**
- **Destrucción de datos**



Taint Mode establece una política de seguridad

- Los datos recibidos desde un el exterior se consideran **no confiables** (o *manchados*)
- Los datos no confiables se pueden volver confiables (o *sin manchas*) mediante un **proceso de sanitización**.
- Los datos no confiables no deben llegar a un **sumidero sensible**.

Mecanismos a implementar

- Marcar **entradas no confiables**, **funciones limpiadoras** y **sumideros sensibles**.
 - **Desmachar** los datos al sanitizarlos
 - **Detectar** cuando un dato manchado alcanza un sumidero sensible
 - **Propagar** información sobre las manchas
- 

Propagación de manchas

```
a # manchada
```

```
b # limpia
```

```
c = a + b # c está manchada
```

```
a * 8
```

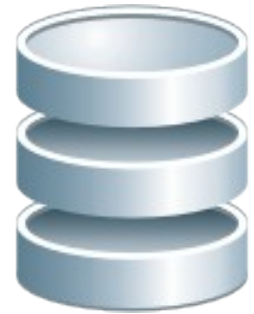
```
a[3:10]
```

```
"esta %s limpia?" % a
```

```
a.upper()
```

Diferentes tipos de ataques

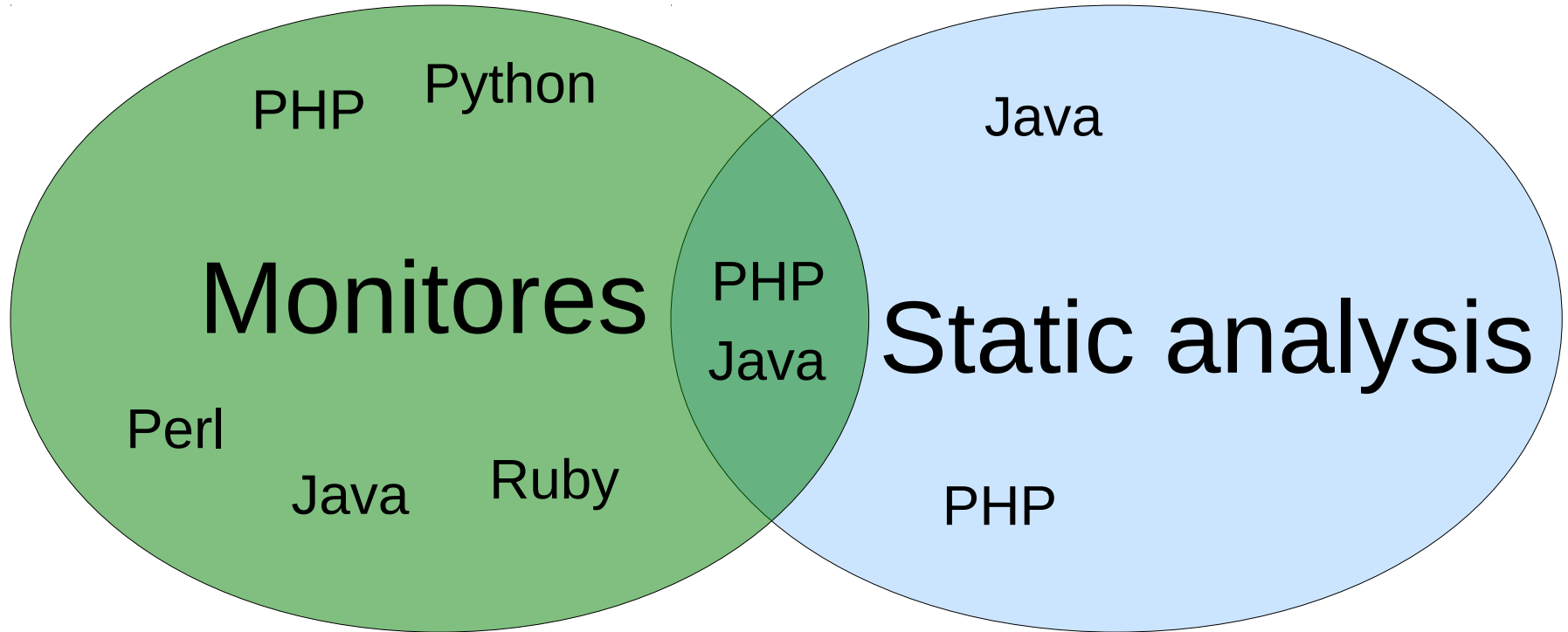
"42 or 1=1"



**"<script>
alert('hola')
</script>"**



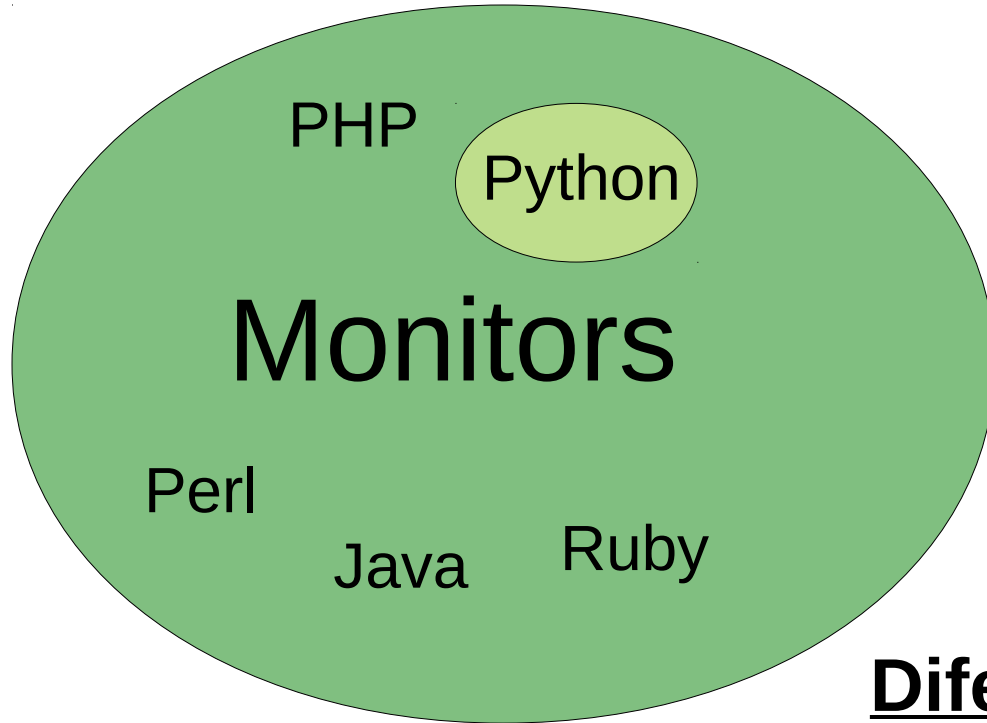
Taint Analysis



- + Menos falsas alarmas que SA
- Overhead
- Modificación del intérprete

- + Sin overhead
- + No modifican el intérprete
- Más falsas alarmas que M

Nuestra implementación



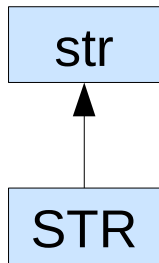
- + Less false alarm than SA
- Overhead

Diferencia con otros [18]

- Modificación del intérprete
- Solo strings
- Atributo binario: si o no
- + SIN cambios en el código

¿Cómo funciona la biblioteca?

```
STR = taint_class(str)
```



Tipos built-in sobrecargados automáticamente

XSS, SQLI

taints

"a"

```
c = a + b  
STR = STR + str  
STR = STR.__add__
```

Funciones built-in sobrecargadas automáticamente

```
c = a.upper()  
STR = STR.upper
```

```
c = len(a)  
INT = len(STR)
```

API

- Fuentes no confiables

```
from web import input
input = untrusted(input)
```

```
@untrusted
def user_function():
    ...
```

API

- Sumideros sensibles

```
db.select = ssink(SQLI)(db.select)
```

```
@ssink(OSI)
```

```
def user_function(cmd):
```

```
    ...
```

API

- Funciones limpiadoras

```
sanitize = cleaner(SQLI)(sanitize)
```

```
@cleaner(OSI)  
def user_function(cmd):  
    ...
```

Conclusiones

- **Es posible escribir una biblioteca liviana (300 LOC) de taint analysis en/para Python**
- No se necesitó modificar el intérprete

¿Por dónde seguimos?

- Google App Engine
 - <http://codereview.appspot.com/1914047/>
- Estoy preparando una tesis de maestría sobre este tema
- Otras personas
 - Phd. Alejandro Russo (Chalmers)
 - Emmanuel Herrmann (Estudiante UTN FRSF)



Más información

- Paper: **A Taint Mode for Python via a library** (EN)
- Artículo en PET 1
 - <http://revista.python.org.ar/1/html/taint.html> (ES)
 - <http://revista.python.org.ar/1/html-en/taint.html> (EN)
- <http://www.cse.chalmers.se/~russo/juanjo.htm>
- **<http://www.juanjoconti.com.ar/taint/>**
- Código fuente
 - svn co <http://svn.juanjoconti.com.ar/dyntaint/trunk/>