Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities

By Nenad Jovanovic, Christopher Kruegel, Engin Kirda

Presented for EECS 583 by Colin Nizielski, Lloyd Shatkin, Nick Adams

Outline

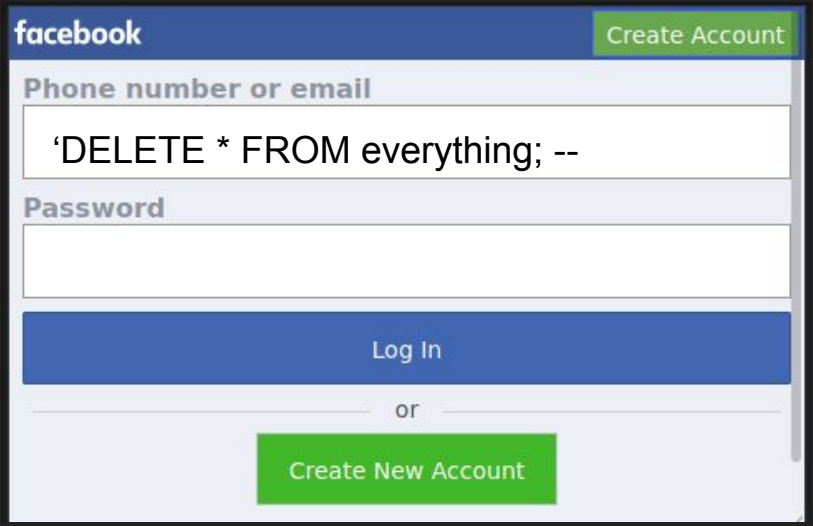
1. Problem Introduction
2. Data Flow Analysis
3. Taint Analysis
4. Empirical Results

Motivation

Why care about website vulnerabilities?

Taint-style Web Vulnerabilities

- All user input is considered **tainted data**
- Vulnerable parts of a program are **sensitive sinks**
- Tainted data can be **sanitized** for safe use
- Example attacks exploiting tainted data
  - Path Traversal
  - Command Injection
  - SQL Injection
  - Cross Site Scripting



Cross Site Scripting (XSS)

- Allows malicious JavaScript to be run in users' browsers
- Easy to avoid but easy to miss
- One possible scenario
    1. Attacker submits malicious content containing executable code
    2. Normal user views content and unknowing runs malicious code
    3. Code sends user information back to the attacker

How to prevent XSS attacks

- Understand the tainted inputs to a web service
  - GET/POST requests
  - Cookies
  - Databases of user generated content
- Run sanitizing routines on data
  - Remove any executable code
  - Escape special characters
- Confirm anything returned to the user is untainted

What is Pixy?

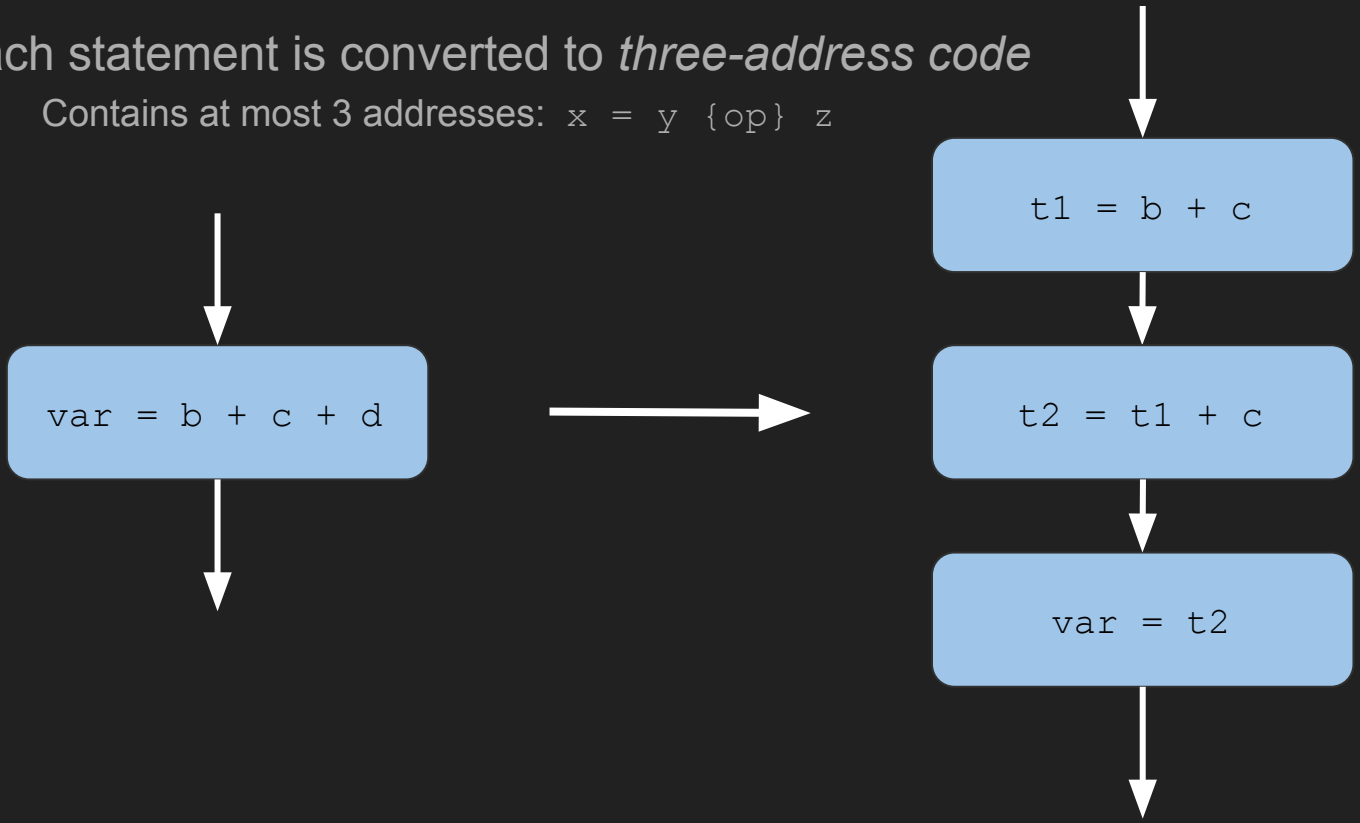
- An open source static, data flow analysis tool for PHP
- Focused on detecting taint-style vulnerabilities, specially XSS
- Multi step analysis:
  - Constants and literals analysis
  - Aliased value analysis
  - Taint analysis

Outline

1. Problem Introduction
2. **Data Flow Analysis**
3. Taint Analysis
4. Empirical Results

Intermediate Representation: P-Tac

- Each statement is converted to *three-address code*
  - Contains at most 3 addresses: `x = y {op} z`



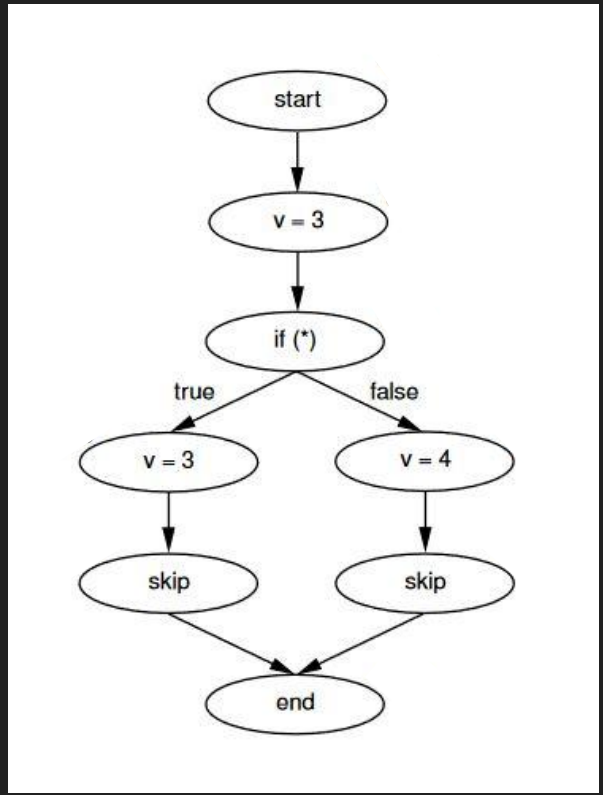
P-Tac: CFG Nodes

- P-Tac condenses the potentially infinite description of statements

CFG Node	Shape / Description
Simple Assignment	{var} = {place}
Unary Assignment	{var} = {op} {place}
Binary Assignment	{var} = {place} {op} {place}
Array Assignment	{var} = array()
Reference Assignment	{var} &= {var}
Unset	unset({var})
Global	global {var}
Call Preparation	A call node's predecessor.
Call	Represents a function call.
Call Return	A call node's successor.

Literal Analysis

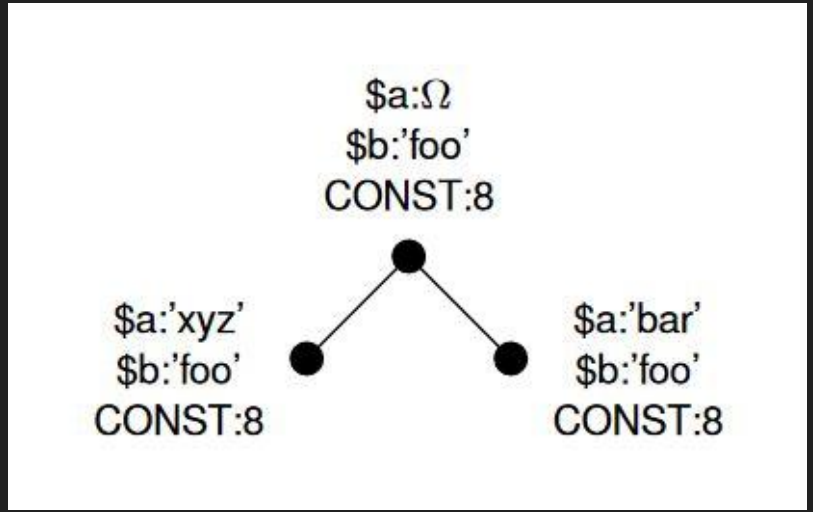
- **Purpose**: Find the literal value that a variable or constant can hold
- Utilizes Control flow graphs of atomic operations
- Iterative analysis that is flow and context sensitive as well as interprocedural



Literal Analysis: Carrier Lattice

- Provides mappings for all variables and constants at any execution point

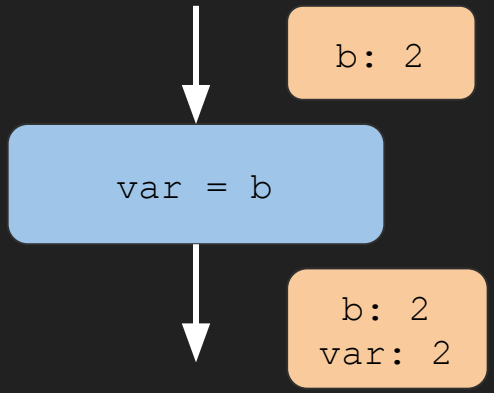
- Ω signifies unknown value



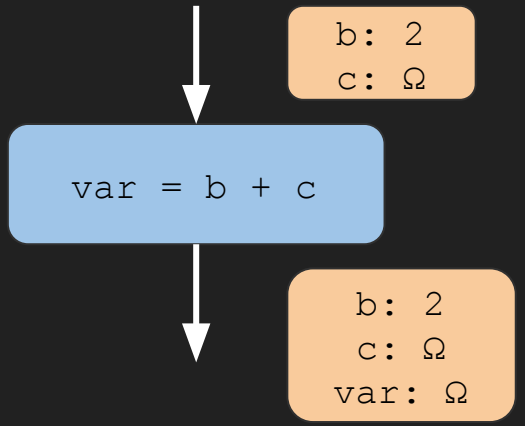
Literal Analysis: Transfer Functions

- Transfer functions define how information is affected by each CFG node

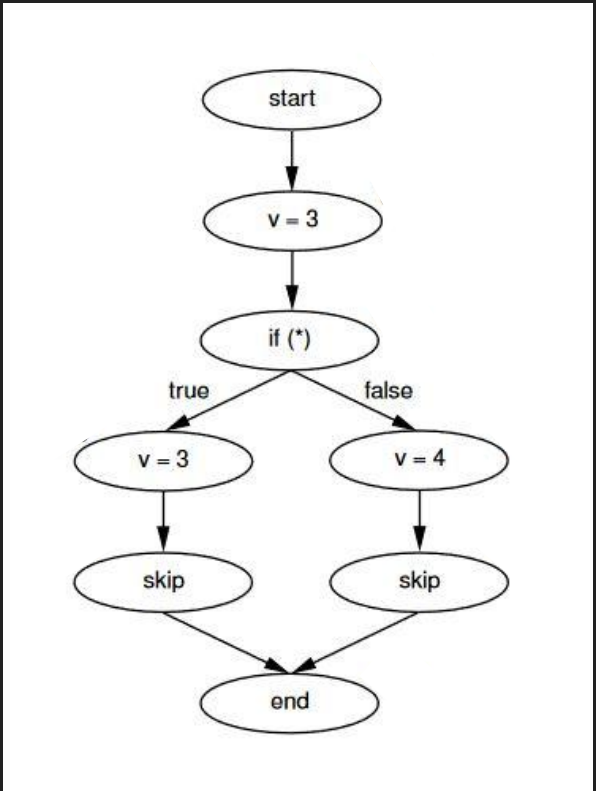
Simple Assignment:



Binary Assignment:



Literal Analysis in Action



- Variable `v` is initially unknown
- Simple assignment transfer function updates carrier lattice
- Simple assignment transfer function updates carrier lattice
- End node merges different parent lattices for variable `v` into unknown (Ω)

Literal Analysis: Aliases

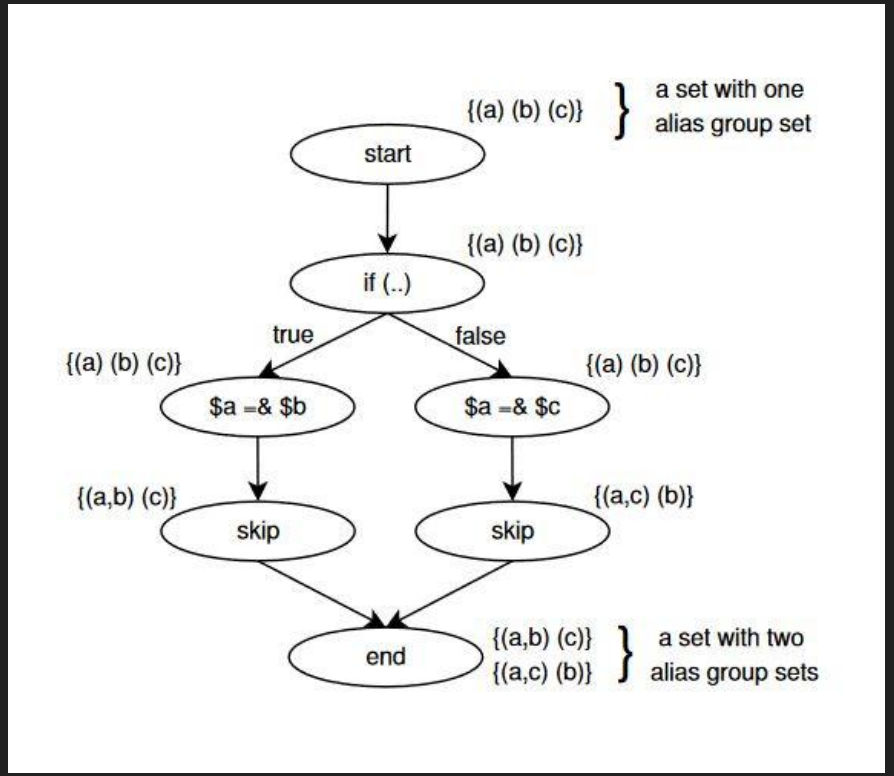
- Currently literal analysis would not be able to detect that instruction 4 also affects variable b

```
1: $a = 1;
2: $b = 2;
3: $b =& $a;   // $b == 1
4: $a = 3;     // $a == 3, $b == 3
```

Alias Analysis

- Carrier Lattice structures now include alias sets

- Conservative All-Path merge



Outline

1. Problem Introduction
2. Data Flow Analysis
3. **Taint Analysis**
4. Empirical Results

Taint Analysis - Basics

- A variable is tainted if it can hold malicious values
  - Originates from user input
- For carrier lattice, map to values *tainted* and *untainted*
- Conservative approach
  - “Might be tainted”
  - “Must be untainted”

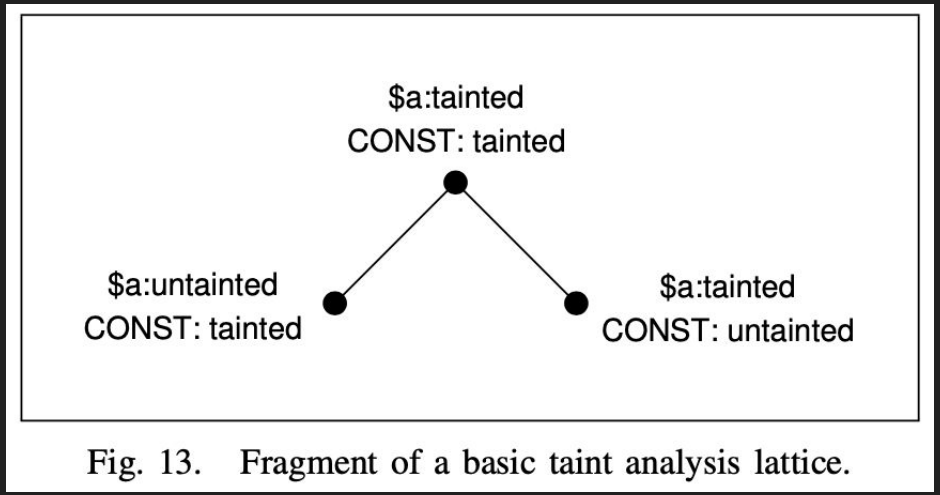


Fig. 13. Fragment of a basic taint analysis lattice.

Taint Analysis - Clean Array Flag

- Literal analysis treats non-literal array elements pessimistically
  - $a[$i]
- Leads to false positives in taint analysis
- Track *Clean Array Flag (*CA Flag)
  - Overrides taint analysis declaring non-literal elements as tainted

```
1: $a = <user input>;
2: // $a[1] can be controlled by an attacker
3:
4: $a = array();
5: // now $a[1] is no longer controlled by
6: // an attacker
```

Fig. 14. Untainting with “array”.

Taint Analysis - Transfer Functions

- Same process as literal analysis except the addition of CA Flag

Left Variable	Literal Analysis	Taint Analysis
Not an array element and not known as array ("normal variable").	strong update for must-aliases, weak update for may-aliases	strong update (taint, CA flag) for must-aliases, weak update (taint, CA flag) for may-aliases
Array, but not an array element.	strong overlap	target.caFlag = source.caFlag; strong overlap (taint)
Array element (and maybe an array) without non-literal indices.	strong overlap	target.root.caFlag ⊔= source.caFlag; strong overlap (taint)
Array element (and maybe an array) with non-literal indices.	weak overlap for all MI variables	target.root.caFlag ⊔= source.caFlag; weak overlap (taint) for all MI variables

TABLE II

ACTIONS PERFORMED BY LITERAL ANALYSIS AND TAINT ANALYSIS FOR SIMPLE ASSIGNMENT NODES DEPENDING ON THE LEFT-HAND VARIABLE.

Taint Analysis - Limitations

- Does not support object oriented features of PHP
  - Treated optimistically - malicious data can never arise
- Files included with “include” are not scanned automatically
  - File inclusions in PHP are dynamic
  - Names of included files can be constructed at run-time and can even return values
  - Hard to analyze, so treated pessimistically

Outline

1. Problem Introduction
2. Data Flow Analysis
3. Taint Analysis
4. **Empirical Results**

Empirical Results

Method:

- Pixy was run on 6 open-source PHP programs
- Manually resolved “include” relationships by simply providing function definitions
- Each program was analyzed in less than 1 minute
  - 3.0 GHz Pentium 4 processor with 1 GB RAM

Program	File	LOC	Variables	Vulnerabilities	FP's	Advisories
PhpNuke 6.9	Reviews Module	8409	3113	15	5	BugTraq: 10493, 10524, 365368
	YourAccount Module	9070	3452	9	25	BugTraq: 13007, 394971, 394867, 321324
PhpMyAdmin 2.6.0-pl2	select_server.lib.php	89	23	9	0	PMASA-2005-01
Gallery 1.3.3	search.php	1810	530	2	1	BugTraq: 348514
	login.php	1719	488	1	0	BugTraq: 8039
	Totals	21097	7606	36	31	

TABLE III

KNOWN VULNERABILITIES DISCOVERED BY PIXY.

Program	File	LOC	Variables	Vulnerabilities	FP's	Advisories
Simple PHP Blog 0.4.5	preview_cgi.php	6938	2342	3	5	TUVSA-0511-001, BugTraq 415463
	preview_static_cgi.php	6883	2316	4	4	
	colors.php	6971	2313	1	6	
Serendipity 0.8.4	personal.inc.php	6588	2305	2	1	TUVSA-0509-001, BugTraq 412023
Yapig 0.95b	view.php	5128	1302	5	0	TUVSA-0510-001, BugTraq 413255
	Totals	29508	10578	15	16	

TABLE IV

UNKNOWN VULNERABILITIES DISCOVERED BY PIXY.

False Positives (47 Total)

- **14 caused by dynamically initialized global variables in included files**
  - **Can get better by automatically processing “include” files**

False Positives (47 Total)

- 14 caused by dynamically initialized global variables in included files
  - Can get better by automatically processing “include” files
- **13 caused by file reads**
  - **Can be improved by tracking files for which an attacker could inject tainted values**

False Positives (47 Total)

- 14 caused by dynamically initialized global variables in included files
  - Can get better by automatically processing “include” files
- 13 caused by file reads
  - Can be improved by tracking files for which an attacker could inject tainted values
- **7 caused by global arrays**
  - **Due to not covering alias relationships for arrays**

False Positives (47 Total)

- 14 caused by dynamically initialized global variables in included files
  - Can get better by automatically processing “include” files
- 13 caused by file reads
  - Can be improved by tracking files for which an attacker could inject tainted values
- 7 caused by global arrays
  - Due to not covering alias relationships for arrays
- **Others due to more complex reasons with no solution presented**

False Positives (47 Total)

- 14 caused by dynamically initialized global variables in included files
  - Can get better by automatically processing “include” files
- 13 caused by file reads
  - Can be improved by tracking files for which an attacker could inject tainted values
- 7 caused by global arrays
  - Due to not covering alias relationships for arrays
- **Others due to more complex reasons with no solution presented**

Questions?

Extra Slides: Arrays

- Each index referenced is stored in an array tree

- Allows for nested arrays as well