

Software Security

Taint analysis

Alessandra Gorla



Buffer overflow vulnerability

```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

Code still runs; user now 'authenticated'

M e ! \0					
A	u	t	h	4d 65 21 00	%ebp %eip &arg1
buffer			authenticated		

Injectons

- Injection attacks trick an application into including unintended commands in the data sent to an interpreter.
- Interpreters
 - Interpret strings as commands.
 - Ex: SQL, shell (cmd.exe, bash), LDAP, XPath
- Key Idea
 - Input data from the application is executed as code by the interpreter.

SQL injection

```
$link = mysql_connect($DB_HOST, $DB_USERNAME, $DB_PASSWORD) or die ("Couldn't connect: " .  
mysql_error());
```

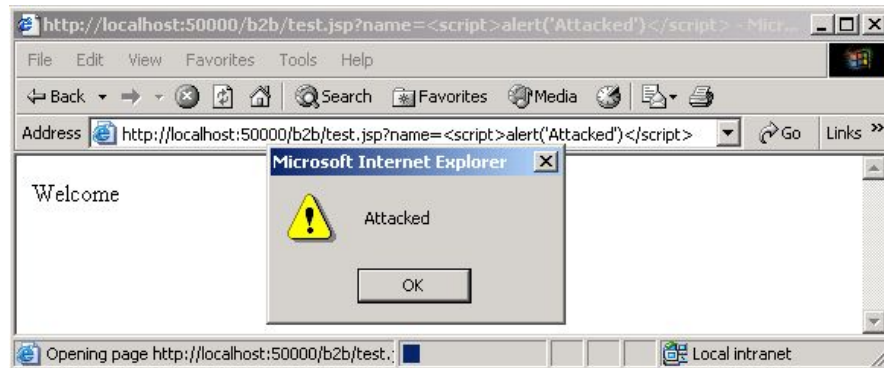
```
mysql_select_db($DB_DATABASE);
```

```
$query = "select count(*) from users where username = '$username' and password = '$password' ";
```

```
$result = mysql_query($query);
```

A simple example

`http://myserver.com/welcome.jsp?name=<script>alert("Attacked")</script>`

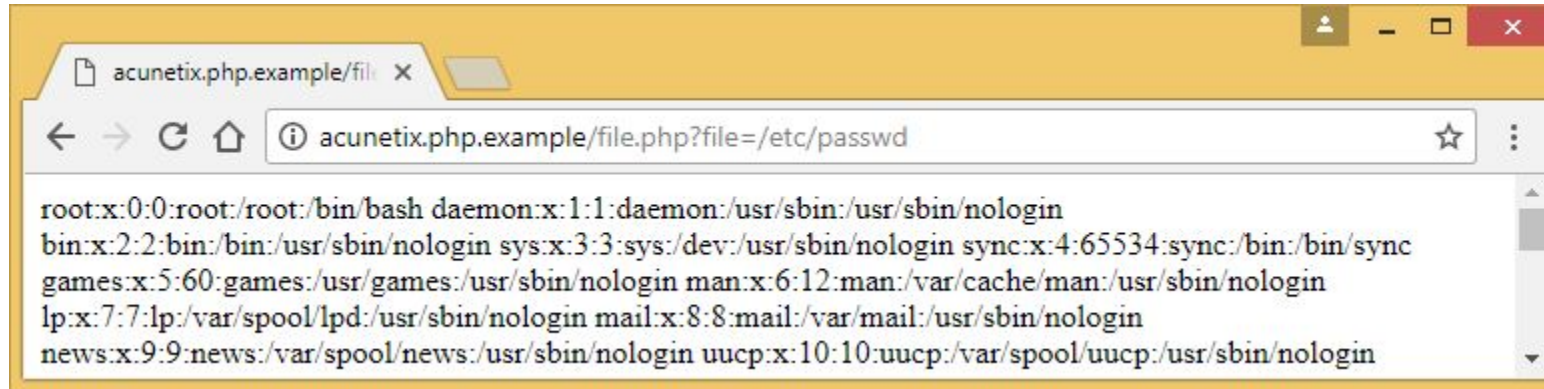


```
<HTML>
<Body>
Welcome <script>alert("Attacked")</script>
</Body>
</HTML>
```

Directory traversal

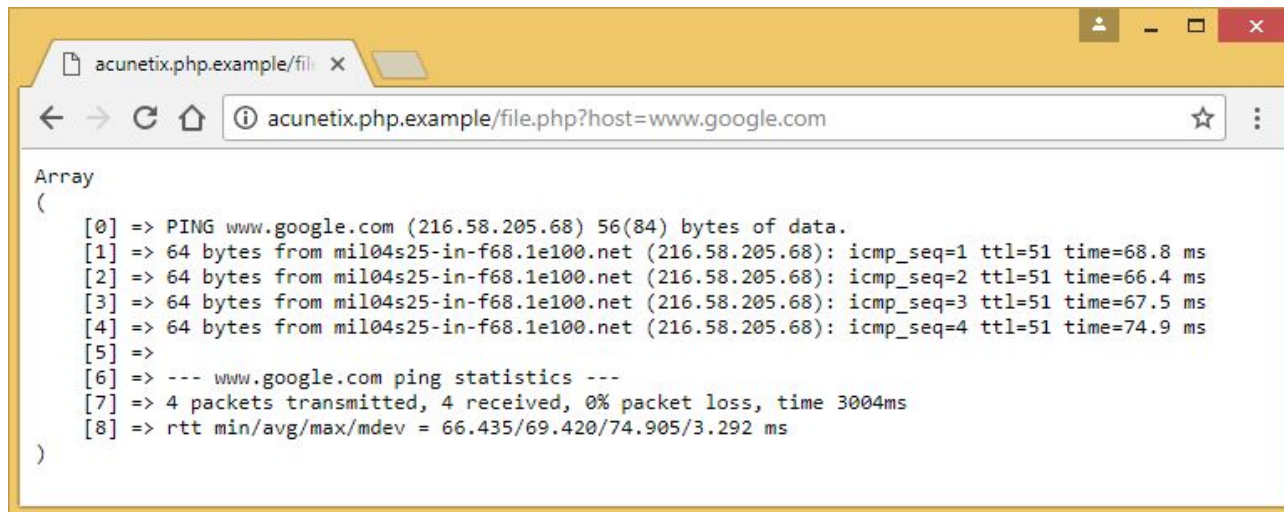
```
$file = $_GET['file'];
```

```
include($file);
```

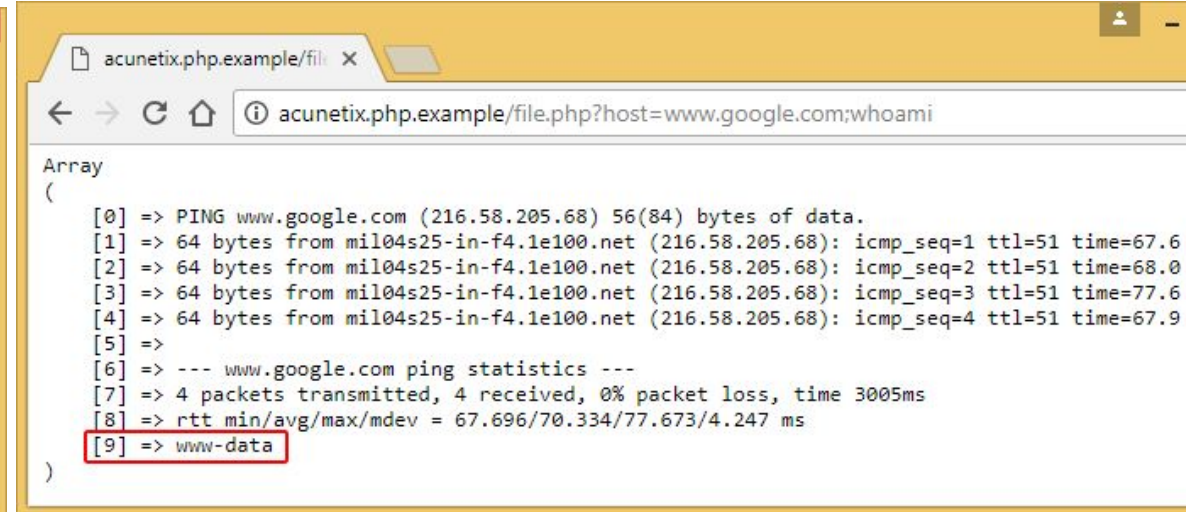


Command injection

```
exec("ping -c 4 " . $_GET['host'], $output);  
echo "<pre>";  
print_r($output);  
echo "</pre>";
```



```
Array  
(  
    [0] => PING www.google.com (216.58.205.68) 56(84) bytes of data.  
    [1] => 64 bytes from mil04s25-in-f68.1e100.net (216.58.205.68): icmp_seq=1 ttl=51 time=68.8 ms  
    [2] => 64 bytes from mil04s25-in-f68.1e100.net (216.58.205.68): icmp_seq=2 ttl=51 time=66.4 ms  
    [3] => 64 bytes from mil04s25-in-f68.1e100.net (216.58.205.68): icmp_seq=3 ttl=51 time=67.5 ms  
    [4] => 64 bytes from mil04s25-in-f68.1e100.net (216.58.205.68): icmp_seq=4 ttl=51 time=74.9 ms  
    [5] =>  
    [6] => --- www.google.com ping statistics ---  
    [7] => 4 packets transmitted, 4 received, 0% packet loss, time 3004ms  
    [8] => rtt min/avg/max/mdev = 66.435/69.420/74.905/3.292 ms  
)
```



```
Array  
(  
    [0] => PING www.google.com (216.58.205.68) 56(84) bytes of data.  
    [1] => 64 bytes from mil04s25-in-f4.1e100.net (216.58.205.68): icmp_seq=1 ttl=51 time=67.6  
    [2] => 64 bytes from mil04s25-in-f4.1e100.net (216.58.205.68): icmp_seq=2 ttl=51 time=68.0  
    [3] => 64 bytes from mil04s25-in-f4.1e100.net (216.58.205.68): icmp_seq=3 ttl=51 time=77.6  
    [4] => 64 bytes from mil04s25-in-f4.1e100.net (216.58.205.68): icmp_seq=4 ttl=51 time=67.9  
    [5] =>  
    [6] => --- www.google.com ping statistics ---  
    [7] => 4 packets transmitted, 4 received, 0% packet loss, time 3005ms  
    [8] => rtt min/avg/max/mdev = 67.696/70.334/77.673/4.247 ms  
    [9] => www-data  
)
```

Common issue?

Input from user is used directly *without any sanitisation*

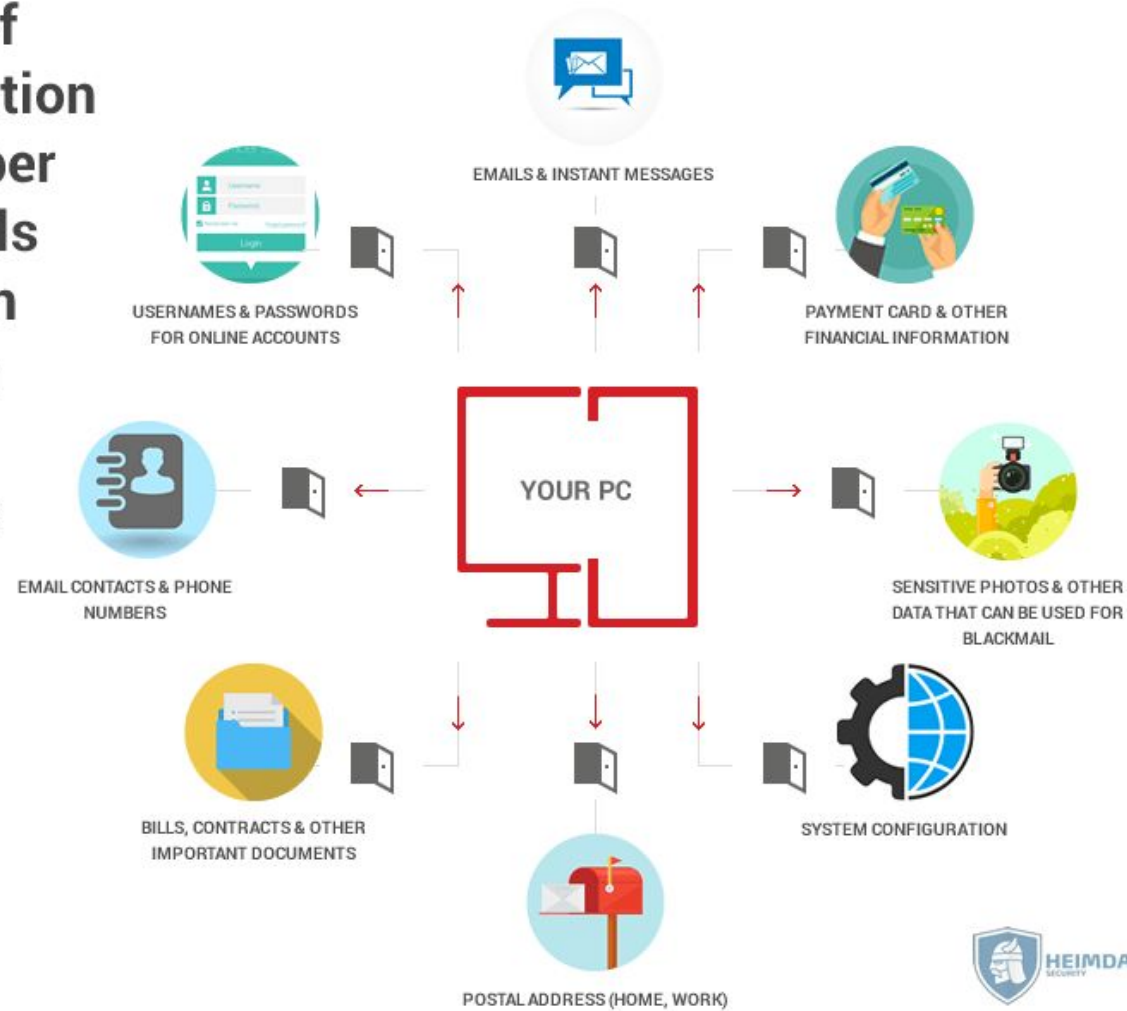
Can we automatically find these issues?

YES! With taint analysis

- Information flow analysis.
- Used in the security domain.
 - Tracking how user input flows to critical statements (e.g. sql execution statements)
 - Tracking how private information flows through the program and if it is leaked to public observers.

Information leak

Types of information that cyber criminals can gain through data leakage



Example

```
1. input = get_input();  
2. tmp = "select.." + input;  
3. query(tmp);  
4. send_internet(tmp);
```

Simple rules

Untainted + **Untainted** = **Untainted**

Untainted + **Tainted** = **Tainted**

Tainted + **Tainted** = **Tainted**

Example

```
1. input = get_input() ;  
2. tmp = "select.." + input ;  
3. query(tmp) ;  
4. send_internet(tmp) ;
```



possible SQL injection
possible sensitive information
leaked

Terminology

- Sources
 - Private data of interest / user inputs
- Sinks
 - Locations of interest / critical statements
 - Check taints of incoming information
 - Determines if there is a leak in the program/ if there is a possible injection

Example

```
1. input = Source();  
2. tmp = "select ..." + input;  
3. Sink(tmp);  
4. Sink(tmp);
```

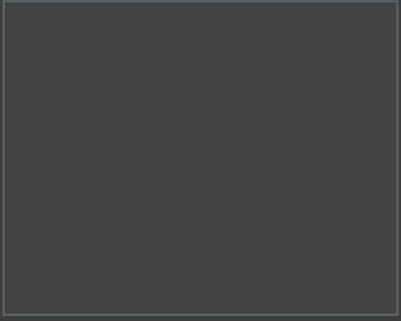


possible SQL injection
possible sensitive information leak

Example

```
1. input = Source();  
2. tmp = "select ..." + input;  
3. tmp = encode(tmp)  
4. Sink(tmp);
```





Dynamic Taint Analysis

Dynamic Taint Analysis

Track what are the taints that are influencing the values of the program

```
1. x = get_input();  
2. y = 1;  
3. z = x;  
4. w = y + z;  
5. exec("SELECT ... "+w);
```

Example

```
1. x = Source(0);  
2. y = 1;  
3. z = x;  
4. w = y + z;  
5. Sink(w);
```

x  **0** → T



Is there a warning?

```
1.      x  =  Source (0) ;
2.      y  =  x;
3.      if (y == 0) {
4.          z  =  2
5.      }
6.      else {
7.          z  =  1
8.      }
9.      Sink (z) ;
```

Implicit flows

- Tainted data affects the value of another variable indirectly.
- Needed for sound analysis.

Implicit flows

```
1.      x  =  Source (0) ;  
2.      y  =  x;          ← Explicit information flow  
3.      if (y == 0) {  
4.          z  =  2      ← Implicit information flow  
5.      }  
6.      else {  
7.          z  =  1      ← Implicit information flow  
8.      }  
9.      Sink (z) ;
```

The diagram illustrates information flow in a code snippet. An arrow points from the expression `Source (0)` in line 1 to the assignment `y = x;` in line 2, labeled "Explicit information flow". Two arrows point from the `if` and `else` branches to the variable `z` in lines 4 and 7, labeled "Implicit information flow".

Implicit flows

```
1.      x  =  Source (0) ;
2.      y  =  x;
3.      if (y == 0) {
4.          z  =  2
5.      }
6.      else {
7.          z  =  1
8.      }
9.      Sink (z) ;
```

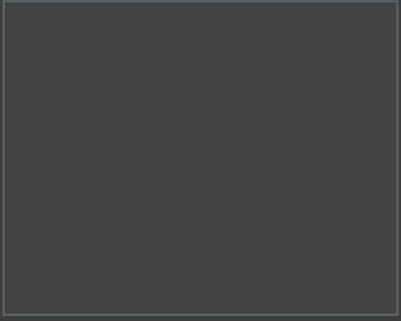
x  **0** \rightarrow **T**



Limits of Dynamic Analysis

Results are input dependent.

Implicit flows needed for sound analysis, but difficult to track.



Static Taint Analysis

Static Taint Analysis

- Track, at each instruction, what are the taints that are influencing the variables of the program.

Example

1. `x = Source();`

2. `y = 1;`

3. `z = x;`

4. `w = y + z;`

5. `Sink(w);`

$x \rightarrow T$

$x \rightarrow T$

$x \rightarrow T, z \rightarrow T$

$x \rightarrow T, z \rightarrow T, w \rightarrow T$

$x \rightarrow T, z \rightarrow T, w \rightarrow T$



Dynamic – static

```
1.      x  =  Source ( ) ;  
2.      y  =  x ;  
3.      if ( y == 0 ) {  
4.          z  =  2  
5.      }  
6.      else {  
7.          Sink ( y ) ;  
8.      }
```

Limitations of static analysis

- Do not know what values might cause the leak.
- Overtainting

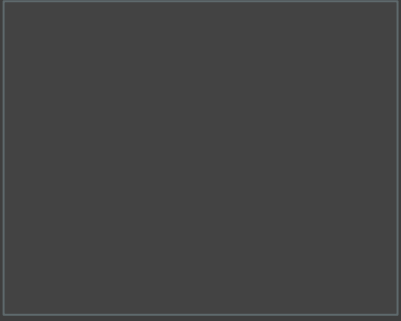
e.g.

```
1. x = Source(args[0]);  
2. Object o = foo();  
3. v = o.equals(x);
```

All implementations of equals should be analyzed!

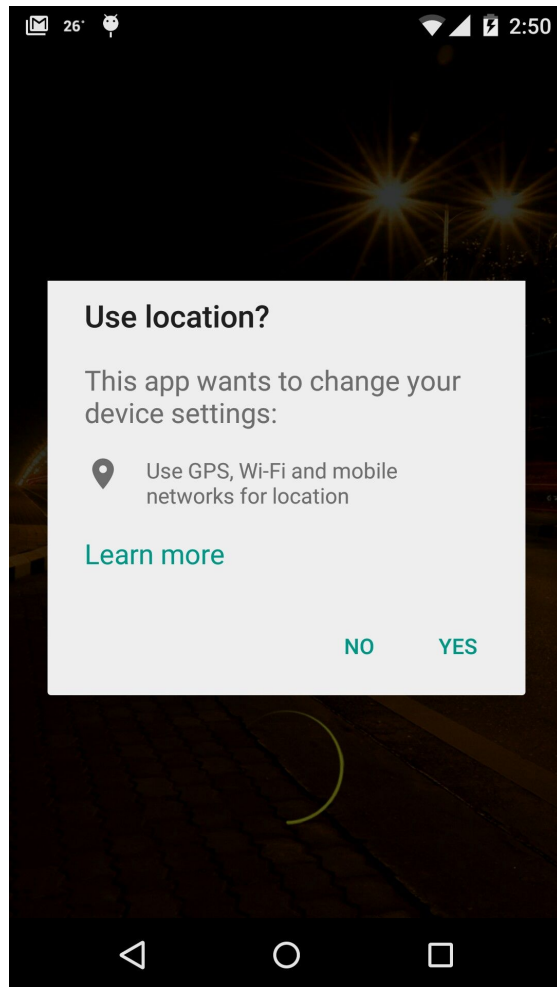
Combining static and dynamic analyses

- Dynamic analysis may miss some information
- Static analysis may report false alarms
- Best solution: Combine them when possible!



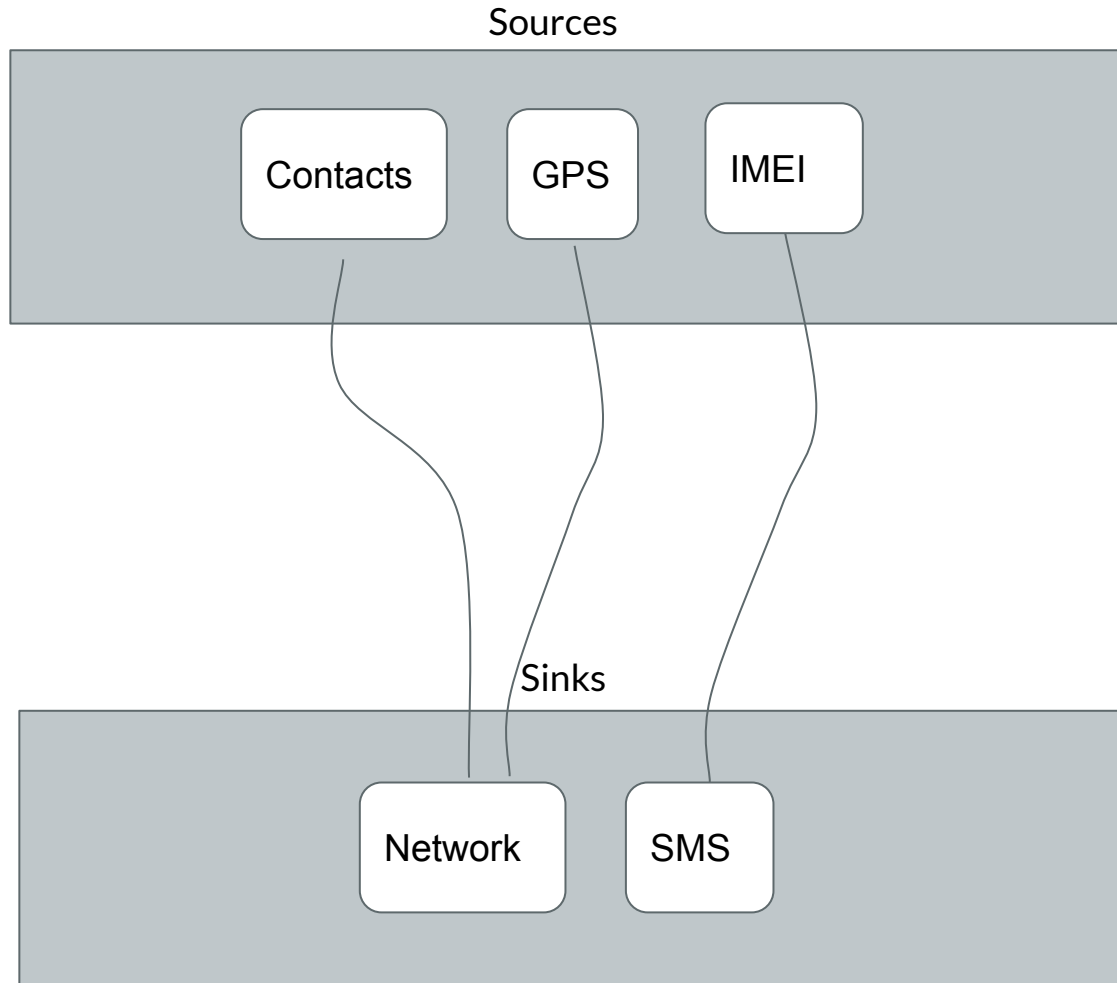
Taint Analysis for Privacy

Taint analysis for mobile apps



- Should I grant the permission?
- What does the app do with this information?
- Does it send my location somewhere?

Taint analysis for mobile apps



Sources: where data comes from
Sinks: where data flows to

How to do that in Android?

```
deviceInfo = AndroidAPI.getDeviceInformation();  
gpsInfo = AndroidAPI.getGPSInformation();
```

...

```
Network.send(deviceInfo.IMEI, gpsInfo.Location);
```