Software Security

Sergey Mechtaev
mechtaev@pku.edu.cn

Peking University

What is Software Security?

- Protecting software from malicious attacks and unauthorized access.
- Ensuring confidentiality, integrity, and availability (CIA triad).

Common Types of Security Vulnerabilities:

- Injection Attacks
- Cross-Site Scripting (XSS)
- Memory Safety Vulnerabilities
- Security Misconfigurations
- ...

SQL Injection

Consider the following code that query database:

```
userName = request.getParameter("user");
statement = “SELECT * FROM users WHERE name = ‘ “  + user + “ ‘; “
executeQuery(statement)
```

SQL Injection

If an attacker enters the following as userName:

```
a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = ‘t
```

The resulting SQL statement will look as follows (deleting users and stealing user data):

```
SELECT * FROM users WHERE name = 'a';DROP TABLE users; SELECT * FROM
userinfo WHERE 't' = 't';
```

Dynamic Taint Analysis

- Track information flow through a program at runtime
- Identify sources of taint – “TaintSeed”
  - Untrusted input
  - Sensitive data
- Taint Policy – “TaintTracker”
  - Propagation of taint
- Identify taint sinks – “TaintAssert”
  - Taint checking
    - Special calls (jump statements, format strings)
    - Outside network

Dynamic Taint Analysis

```
x = get_input( )
…
y = x + 42
…
goto y
```

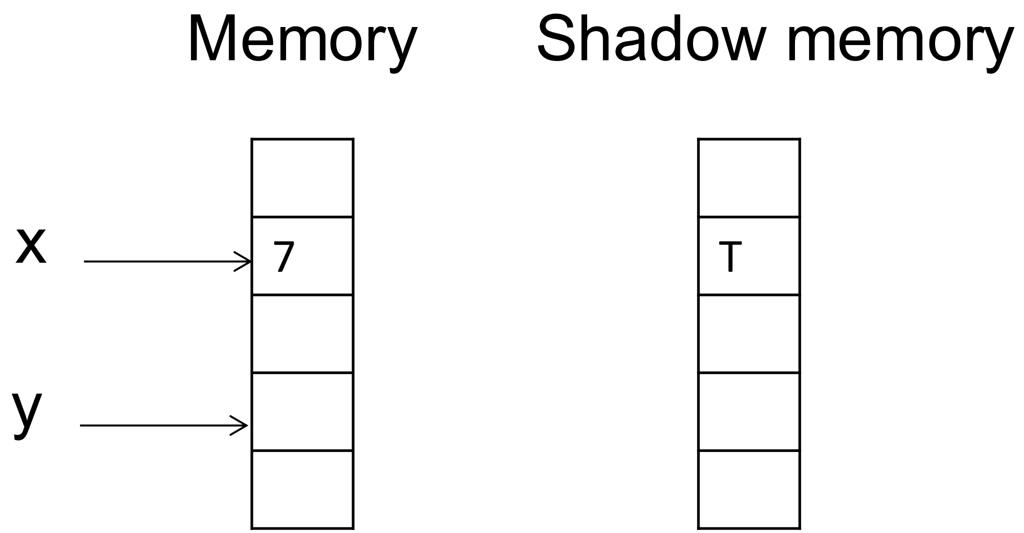
Input is tainted

Dynamic Taint Analysis

```
x = get_input( )
…
y = x + 42
…
goto y
```

– tainted

Shadow Memory

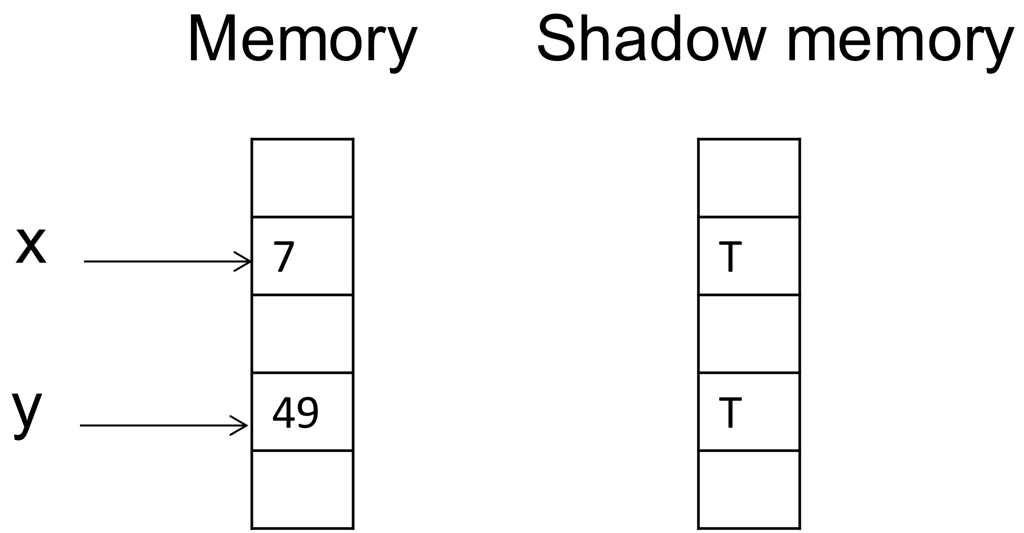


Dynamic Taint Analysis

```
x = get_input( )
…
y = x + 42
…
goto y
```

– tainted

Shadow Memory



Dynamic Taint Analysis

```
x = get_input( )
…
y = x + 42
…
goto y
```

Policy violation detected

– tainted

Issues of Tainting

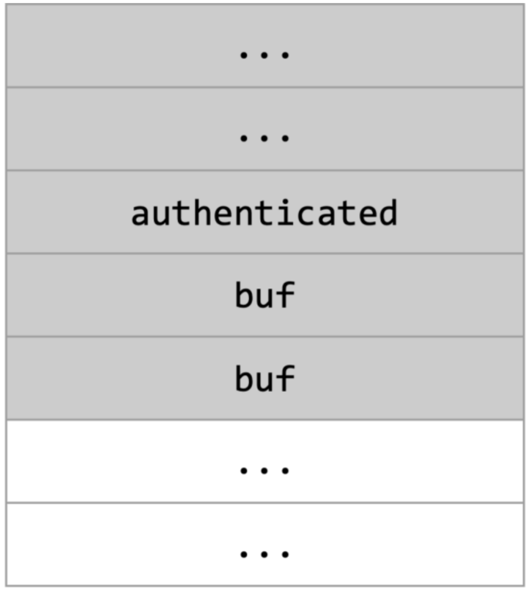
```
x = get_input(   )
…
y = load(x)
…
goto y
```

Not tainting:
table indices can be exploited by attackers

Tainting:
Some applications dispatch based on provided data (e.g. tcpdump)

Memory Safety Vulnerabilities

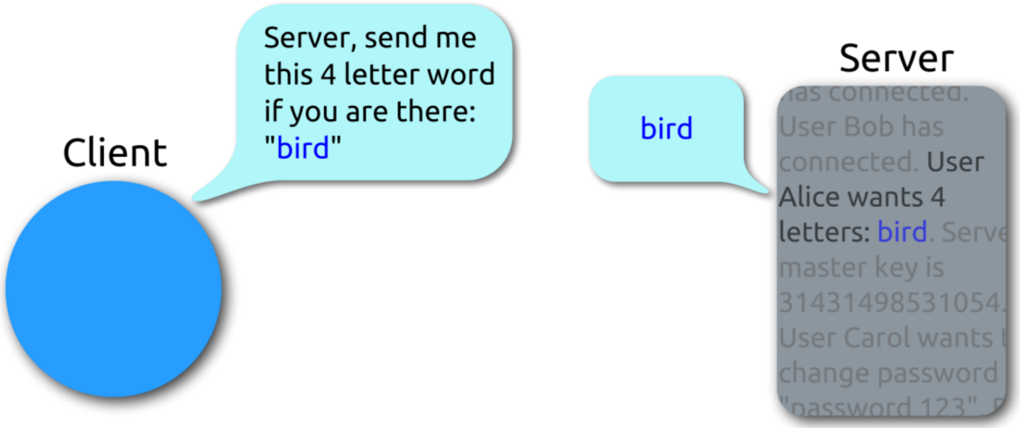
```
char buf[8];
int authenticated = 0;
void vulnerable() {
    gets(buf);
}
```



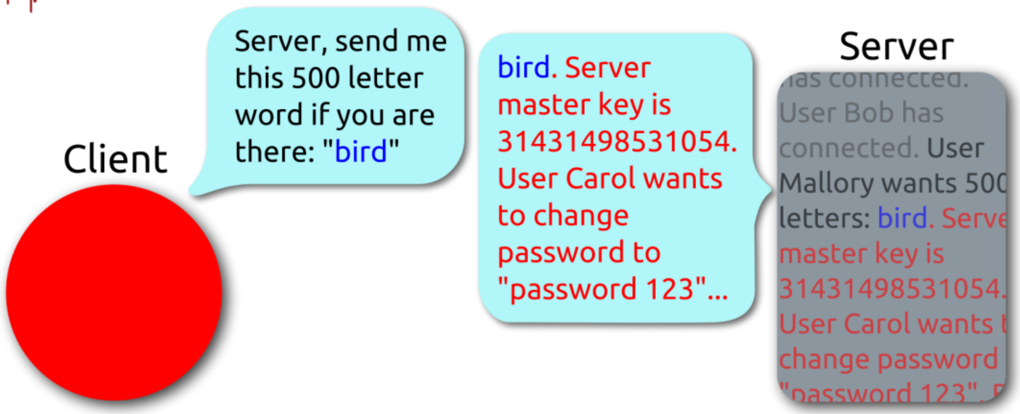
If the attacker can write 9 bytes of data to `buf` (with the 9th byte set to a non-zero value), then this will set the `authenticated` flag to true, and the attacker will be able to gain access.

Heartbleed (Vulnerability in OpenSSL, 2014)

Heartbeat – Normal usage



Heartbeat – Malicious usage



Patch (Added Check)

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0;

if (hbtype == 1) {
    …
```

Aliases

x and y are **aliases** if they point to the same memory cell.

&x – address of x

*x – dereferencing of x

$$[p \coloneqq \&z]^1;$$
$$[z \coloneqq 1]^2;$$
$$[* p \coloneqq 2]^3;$$
$$[print(z)]^4;$$

Does the definition (z,2) reach 4?

When Aliasing Occurs?

- Using pointers
  int *p, i;
  p = &i;
- Call-by-reference, e.g. consider method void m(Object a, Object b)
  m(x,x); // a and b alias in the body of m
  m(x,y); // y and b alias in the body of m
- Array indexing, e.g. int i,j,a[100]
  if $i = j$ then $a[i]$ and $a[j]$ alias

Alias Analysis vs Points-to Analysis

Alias analysis computes a set of pair of variables $\{(x, y)\}$ where x and y may (or must) point to the same memory location.

Points-to analysis computes a relation $points\_to(p, x)$ where p may (or must) point to the location of x.

Example 1: Optimisation with Pointer Analysis

$x = 1;$
$* p = 12;$
$y = x;$

Is the variable x live at the exit of the first statement?

Only if we can determine that p must not point to x.

Then, the program can be optimised into

$* p = 12;$
$y = 1;$

Example 2: Detecting Security Vulnerabilities With Pointer Analysis

```
void copyString(char *input) {
  char buffer[3];
  for (int i=0; i<=3; i++)
    buffer[i] = input[i];
  }
}
```

In C, array references are pointers:

buffer[n] is *(buffer + n)

A pointer analysis can determine that when executing this code, buffer[3] may point to input[3]

buffer[3] is outside of our buffer

Example 3: Memory Management with Pointer Analysis

Rust addresses **memory safety (no dangling pointers, no double-free, etc)** through static analysis.

Rust’s approach: disallow both aliasing and mutation at the same time.

Reasoning: if an object is both aliased and modified, it can cause difficulties. For example, destroying an object with multiple references can create a dangling pointer.

Example 3: Memory Management with Pointer Analysis

A borrower (v1) cannot access the resource after the owner (v) has destroyed it:

```
let v1: &Vec<i32>;
{
   let v = Vec::new();
   v1 = &v;
} //v is dropped here
v1.len(); //error:borrowed value does not live long enough
```

Example 3: Memory Management with Pointer Analysis

Although there could be multiple shared references, there can only be one mutable reference at one time:

```
let mut v:Vec<i32> = Vec::new();
let v1 = &mut v; //first mutable reference
let v2 = &mut v; //second mutable reference
v1.push(1); //error:cannot borrow `v` as mutable more than
once at a time
```

May vs Must Points-to Analysis

A sound **must pointer analysis** will return only those points-to relations that will definitely hold in each possible execution of the program.

A sound **may pointer analysis** reports at least all points-to relations that may occur, i.e. it is an over-approximation.

Andersen’s Points-to Analysis

Lars Ole Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen, 1994

Flow- and context-insensitive analysis.

Represented by a set of rules of the form subset relations.

Constraints

$pt(a)$ is the points-to set of a

	Code	**Constraint**
Referencing	$a := \&b$	$\{b\} \subseteq pt(a)$
Aliasing	$a := b$	$pt(b) \subseteq pt(a)$
Dereferencing read	$a :=* b$	$\{c\} \subseteq pt(b) \Rightarrow pt(c) \subseteq pt(a)$
Dereferencing write	$* a := b$	$\{c\} \subseteq pt(a) \Rightarrow pt(b) \subseteq pt(c)$

Example 1

Program:

$$a \coloneqq \&b;$$
$$c \coloneqq a;$$
$$a \coloneqq \&d;$$
$$e \coloneqq a;$$

- Constraints:

$$\{b\} \subseteq pt(a)$$
$$pt(a) \subseteq pt(c)$$
$$\{d\} \subseteq pt(a)$$
$$pt(a) \subseteq pt(e)$$

- Solution:

$$pt(a) = \{b, d\}$$
$$pt(c) = \{b, d\}$$
$$pt(b) = pt(d) = \emptyset$$
$$pt(e) = \{b, d\}$$

Example 2

Program:

$$a := \&b;$$
$$c := \&d;$$
$$\mathrm{e} := \&a;$$
$$\mathrm{f} := \mathrm{a};$$
$$*\,e := \mathrm{c};$$

Solution:

$$pt(a) = \{b, d\}$$
$$pt(c) = \{d\}$$
$$pt(e) = \{a\}$$
$$pt(f) = \{b, d\}$$

Constraints:

$$\{b\} \subseteq pt(a)$$
$$\{\mathrm{d}\} \subseteq pt(c)$$
$$\{a\} \subseteq pt(e)$$
$$pt(a) \subseteq pt(f)$$
$$\{z\} \subseteq pt(e) \Rightarrow pt(c) \subseteq pt(z)$$

Generated constraint:

$$pt(c) \subseteq pt(a)$$

As Graph Algorithm

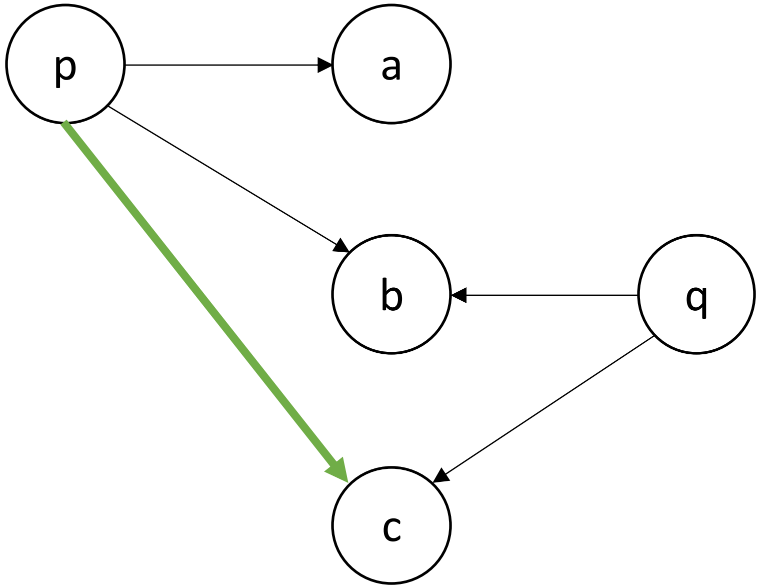
- Can be formalised as a graph transitive closure computation
- Each statement updates the points-to graph if it can creates new points-to relationship

Statement p:=&a

Add an arc from p to a, showing p can possibly point to a:

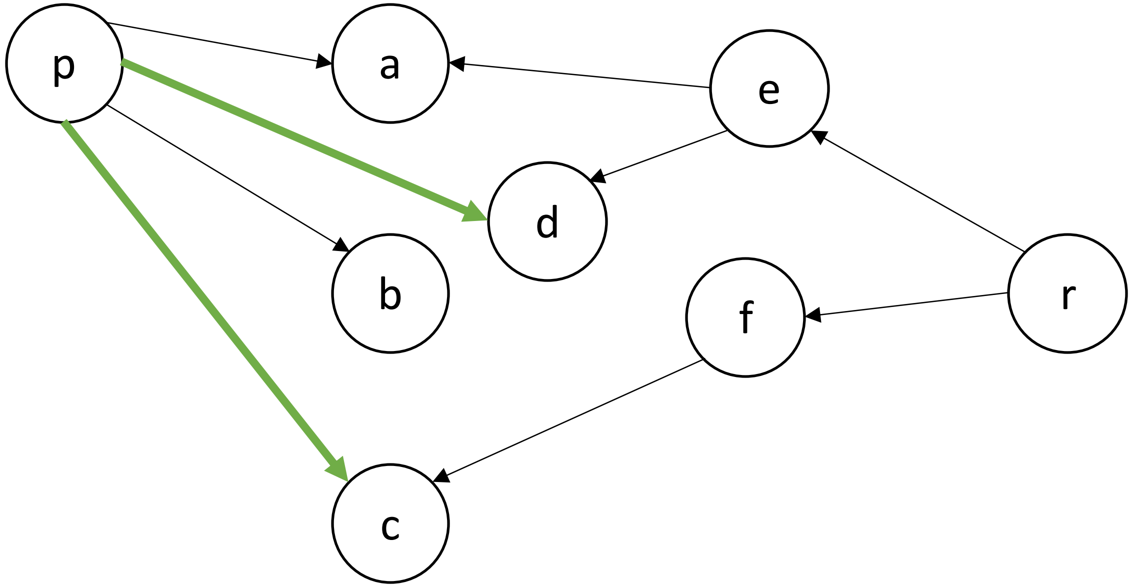
Statement p:=q

Add an arc from p to everything q points to. If new arcs from q are later added, corresponding arcs from p must also be added (iterative fixed point computation):



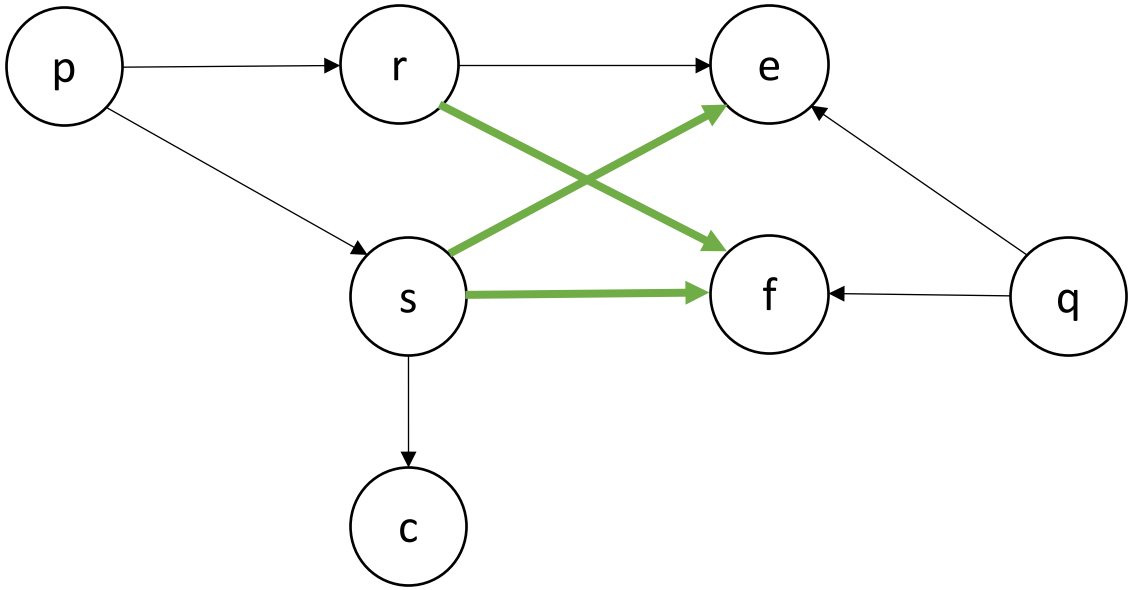
Statement p:=*r

Let S be all the nodes r points to. Let T be all the nodes members of S point to. We add arcs from p to all nodes in T. If later pointer assignments increase S or T, new arcs from p must also be added:



Statement *p = q

Nodes pointed to by p must be linked to all nodes pointed to by q. If later pointer assignments add arcs from p or q, this assignment must be revisited:



Exercise

- Show that Andersen's analysis concludes for this code that D may point to C.
- Argue that for any program that has this set of assignments, no matter which control flow exists between them, D never points to C in any execution.

```
A = &C;
C = &B;
B = &A;
B = A;
*B = C;
D = *A;
```