```java
public class Puzzle13 {

    public static void main(String[] args){
        A b1 = new B();
        A c1 = new C();

        A b2 = b1;
        A c2 = c1;

        // what will get printed?
        b2.print(c2);
    }

    public static class A extends Object {
        public void print(A object) {
            System.out.println("An instance of " + object.getClass().getSimpleName()
                    + " was passed to A's print(A object)");
        }
    }

    public static class B extends A {
        public void print(A object) {
            System.out.println("An instance of " + object.getClass().getSimpleName()
                    + " was passed to B's print(A object)");
        }
    }

    public static class C extends B {
        public void print(A object) {
            System.out.println("An instance of " + object.getClass().getSimpleName()
                    + " was passed to C's print(A object)");
        }
    }

    public static class D extends A {
        public void print(A object) {
            System.out.println("An instance of " + object.getClass().getSimpleName()
                    + " was passed to D's print(A object)");
        }
    }

}
```

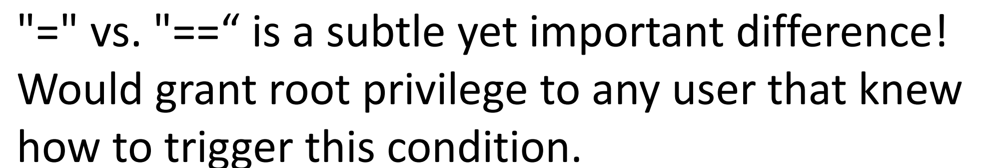
A bug or malware?

```
if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
        retval = -EINVAL;
```

A bug or malware?

- Context: Found in a CVS commit to the Linux Kernel source

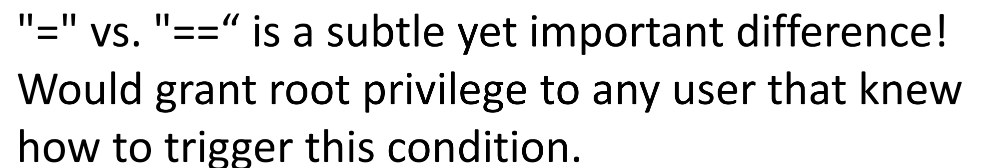
```
if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
        retval = -EINVAL;
```



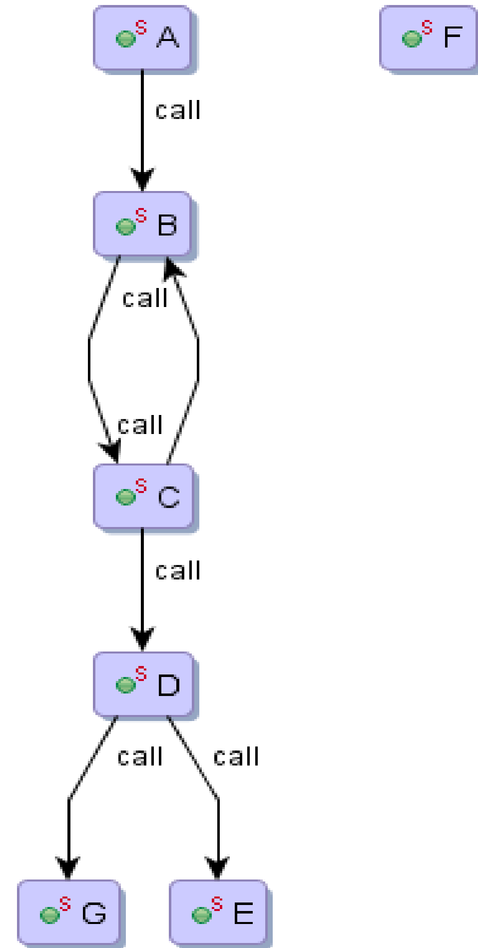
Malware: Linux Backdoor Attempt (2003)

- https://freedom-to-tinker.com/blog/felten/the-linux-backdoor-attempt-of-2003/

```
if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
        retval = -EINVAL;
```



How do we produce a call graph?



Sound and Precise

- We say a call graph is “sound” if it has all the edges that are possible at runtime.
- We say a call graph is “precise” if it does not have edges that do not occur at runtime.
- It’s easy to be sound, but its hard to be sound *and* precise.

Idea 1: Reachability Analysis (RA)

- For Every Method ***m***
  - For Every Callsite
    - Add an edge (if one does not already exist) from ***m*** to any method with the same name as the method called in the callsite

- Refinement: Match methods with the same name AND the same return types and parameter types/counts

- Sound but not precise
  - We end up with a call graph that always has a call edge that *could* happen, but we have a lot of edges that also can not happen.
  - WHY? -> Static Dispatch vs. Dynamic Dispatch
    - Dynamic Dispatches are a problem

Static Dispatch vs. Dynamic Dispatch

- Static Dispatch
  - Resolvable at compile time
  - Includes calls to Constructors and methods marked “static” (ex: main method)
  - Static methods do not require an object instance, they can be called directly

Examples

```
Animal a = new Dog(); // static dispatch to new Dog()
Animal.runSimulation(a); // static dispatch to runSimulation()
```

Static Dispatch vs. Dynamic Dispatch

- Dynamic Dispatch
  - Resolvable at runtime
  - Includes calls to member methods (virtual methods)
  - Requires an object instance, they can be called directly
  - Very common in OO languages such as Java
  - Can be simulated with function pointers

Examples

```
Animal a = new Dog(); // static dispatch to new Dog()
a.toString(); // dynamic dispatch to toString
a.getName(); // dynamic dispatch to dog's getName method
```

Terminology (slight digression)

```
Animal a = new Dog(); // static dispatch to new Dog()
a.toString(); // dynamic dispatch to toString
```

Callsite of String::toString() method (could be overridden in Dog of inherited from Animal or Object)

Receiver object

Declared type (Animal)

Runtime type (Dog)

Idea 2: Class Hierarchy Analysis (CHA)

- Compute the type hierarchy
- For each callsite, if the dispatch is static add the edge like normal, otherwise:
  - Perform RA with the added constraint that the target method must be in either
    - 1) The direct lineage from Object to the receiving object's declared type (in the case that the target method is inherited)
    - 2) The subtype hierarchy of the receiving object's declared type

- Sound and more precise than RA
  - We end up with a call graph that always has a call edge that *could* happen, and we have less edges that can not happen.
  - We can still do better…in terms of precision
  - Checkout the case of: Object o = …; o.toString();
    - We have to add an edge to every toString() method!

Idea 3: Rapid Type Analysis (RTA)

- Summary:
  - Look at the allocation types that were made
  - We can’t have a call to a method in a type that we never had an instance of (in the case of dynamic dispatches)

- Potential Implementation
  - Start with CHA
  - Examine all new allocation types
  - Remove call edges from call graph that point to methods in unallocated types

- Described in detail in “Fast Static Analysis of C++ Virtual Function Calls – IBM”, 1996.
  - More precise than CHA
  - Still sound

Idea 4: Rapid Type Analysis Improvements

- Method Type Analysis (MTA)
  - Idea: Restrict parent method types to types that could be passed through the method's parameters
  - Idea: Consider the statically typed return type of the dynamic dispatch callsite
- Field Type Analysis (FTA)
  - Idea: Consider that a method could write to a global variable, so any allocations reachable by a method are also reachable by a method that reads the same global variable
- Hybrid Type Analysis (XTA)
  - Combines MTA and FTA
  - Precision? More precise than RTA
  - Sound? No…Exceptions are not considered
- Paper: Scalable Propagation-Based Call Graph Construction Algorithms

Idea 5: Variable Type Analysis

- Idea: Track the allocation types to variables the callsites are made on
  - This is a points-to analysis

- Implementation
  - For each new allocation, assign an ID to the new allocation site add each allocation site to the worklist
  - While the worklist is not empty
    - Remove an item from the worklist and propagate its point-to set (set of allocation ids) to every data flow successor (every assignment of the variable to another variable), adding each new variable to the worklist
    - If a callsite is made on the variable, look up the type of the allocation(s) and add a call edge (if parameters are reachable as a result, add the parameters to the worklist)

Call Graph Construction Algorithm Overview