CS165 – Computer Security

Vulnerability discovery – static analysis
Nov 9, 2021

Our Goal

- How to exploit a vulnerability?
- How to find them?

Our Goal

- How to exploit a vulnerability? ✔
- How to find them?
  - Fuzz testing (fuzzing) ✔

Our Goal

- How to exploit a vulnerability? ✔
- How to find them?
  - Fuzz testing (fuzzing) ✔
  - Static analysis

Goal

- Can we build a technique that identifies *all* vulnerabilities?

Goal

- Can we build a technique that identifies *all* vulnerabilities?
  - Turns out that we can: static analysis
  - But, it has its own major limitation
    - Can identify many false positives (not actual vulnerabilities)
  - Can be effective when used carefully

Static Analysis

- Explore all possible executions of a program
  - All possible inputs
  - All possible states

A Form of Testing

- Static analysis is an alternative to dynamic testing
- Dynamic
  - Select concrete inputs
  - Obtain a sequence of states given those inputs
  - Apply many concrete inputs (i.e., run many tests)
- Static
  - Select abstract inputs with common properties
  - Obtain abstract/approximate states created by executing abstract inputs
  - One run

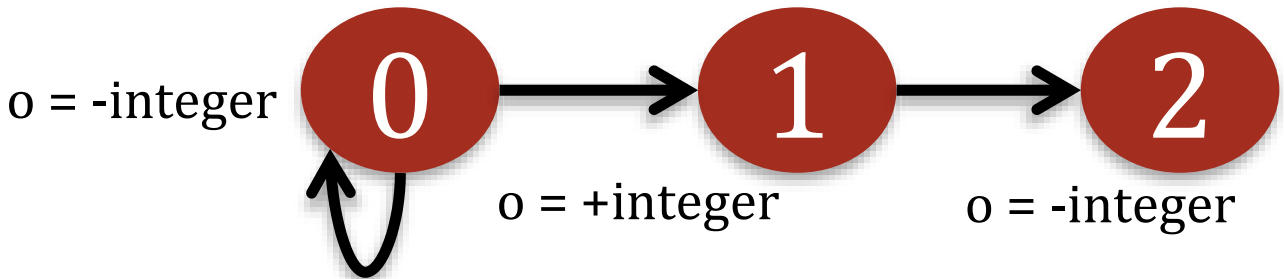
Static Analysis

- Provides an approximation of behavior
- "Run in the aggregate"
  - Rather than executing on ordinary states
  - Finite-sized descriptors representing a collection of states
- "Run in non-standard way"
  - Run in fragments, starting anywhere
- Runtime testing is inherently incomplete, but static analysis can cover all paths
  - But may produce false alarms

Static Analysis

- Provides an approximation of behavior
- "Run in the aggregate"
  - Rather than executing on ordinary states
  - Finite-sized descriptors representing a collection of states

```
int o = -10;
while (true) {
        int i;                          // o negative
        scanf("%d", &i);
        if (i > 0)  continue;           // o still negative
        else if (i < 0)  {
                o = o*i;                // o positive
                o = - o*o;              // o negative
                break;                  // o negative
        }
}
```



Static Analysis

- "Run in non-standard way"
  - Run in fragments, starting anywhere

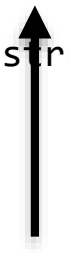
Can start here 

```
int foo(char* fmt) {
    printf(fmt);
}

int main(int argc, char **argv) {
    func1();
    func2();
    …
    foo(argv[1]);
}
```

Static Analysis

- Runtime testing is inherently incomplete, but static analysis can cover all paths
  - But static analysis may produce false alarms

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    cond = foo();
    if(cond)
        strcpy(buf, argv[1]);
    else
        strncpy(buf, argv[1], 64);
}
```



"may" vs. "must" analysis

Static Analysis (project 4)

- Source Code vs. Intermediate Representation

```c
void log_result_advanced(int print)
{
    if(print == 0xefbeadde)
    {
        char filename2[100];
        int uid = getuid();
        // the file needs to be generated at a location where normal users cannot touch
        sprintf(filename2, "uid_%d_crack_advanced", uid);
        printf("file name: %s\n", filename2);
        int fd = open(filename2, O_APPEND | O_CREAT);
        close(fd);
    }
}
```

```
define dso_local void @log_result_advanced(i32 %print) local_unnamed_addr #4 !dbg !50 {
entry:
  %filename2 = alloca [100 x i8], align 16
  call void @llvm.dbg.value(metadata i32 %print, metadata !54, metadata !DIExpression()), !dbg !60
  %cmp = icmp eq i32 %print, -272716322, !dbg !61
  br i1 %cmp, label %if.then, label %if.end, !dbg !62

if.then:                                          ; preds = %entry
  %0 = getelementptr inbounds [100 x i8], [100 x i8]* %filename2, i64 0, i64 0, !dbg !63
  call void @llvm.lifetime.start.p0i8(i64 100, i8* nonnull %0) #8, !dbg !63
  call void @llvm.dbg.declare(metadata [100 x i8]* %filename2, metadata !55, metadata !DIExpression()), !dbg !64
  %call = tail call i32 (...) @getuid() #8, !dbg !65
  call void @llvm.dbg.value(metadata i32 %call, metadata !58, metadata !DIExpression()), !dbg !66
  %call1 = call i32 (i8*, i8*, ...) @sprintf(i8* nonnull %0, i8* nonnull dereferenceable(1) getelementptr inbounds
  %call3 = call i32 (i8*, ...) @printf(i8* nonnull dereferenceable(1) getelementptr inbounds ([15 x i8], [15 x i8]*
  %call5 = call i32 (i8*, i32, ...) @open(i8* nonnull %0, i32 1088) #8, !dbg !69
  call void @llvm.dbg.value(metadata i32 %call5, metadata !59, metadata !DIExpression()), !dbg !66
  %call6 = call i32 (i32, ...) bitcast (i32 (...)* @close to i32 (i32, ...)*)(i32 %call5) #8, !dbg !70
  call void @llvm.lifetime.end.p0i8(i64 100, i8* nonnull %0) #8, !dbg !71
  br label %if.end, !dbg !72

if.end:                                           ; preds = %if.then, %entry
  ret void, !dbg !73
}
```

Static Analysis

- Various properties of a program can be tracked
  - Control flow
  - Data flow
  - Constant propagation
  - Types
- Which ones will expose which vulnerabilities accurately (not too many false positives) requires some finesse

Control Flow Analysis

- Compute the control flow of a program
  - I.e., possible execution paths
- To find an execution path that does not check the return value of a function
  - That is actually run by the program
  - How do we do this?

Control Flow Analysis

- Compute Control Flow
- Function by function – “intraprocedural”
- Program statements of interest
  - Sequences – basic blocks
  - Conditionals – transitions between basic blocks in function
  - Loops – transitions that connect to prior basic blocks
  - Calls – transition to another function
  - Return – transition that completes the function

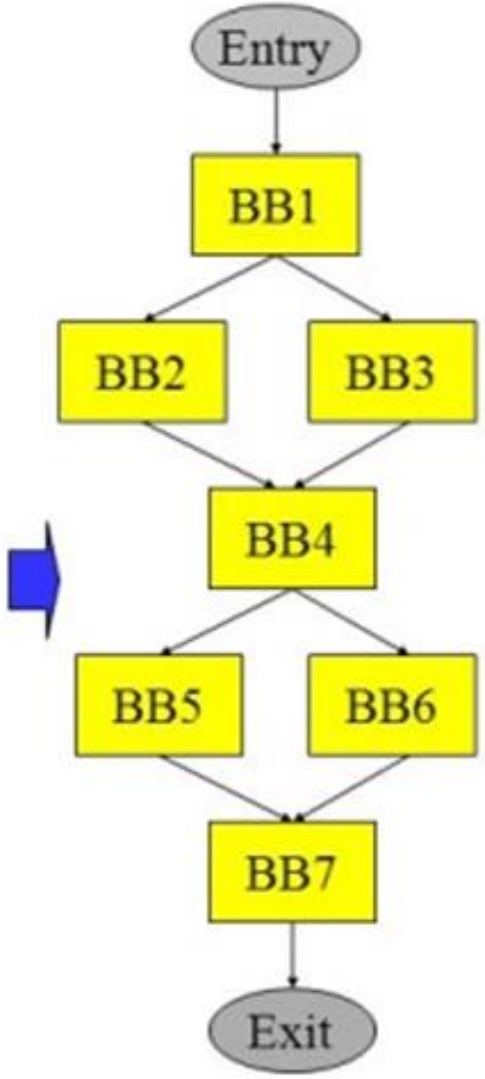
Control Flow Analysis

- Compute Intraprocedural Control Flow

  - Basic block – a sequence of consecutive operations in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end

  - Control Flow Graph – Directed graph, G = (V,E) where each vertex V is a basic block and there is an edge E, v1 (BB1) → v2 (BB2) if BB2 can immediately follow BB1 in some execution sequence

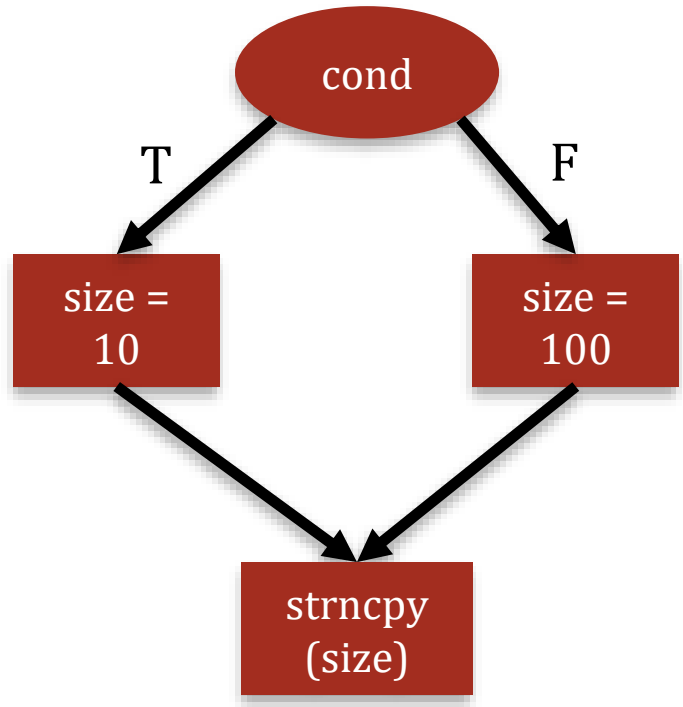
```
x = y+1;
if (c)
   x++;
else
   x--;;
y = z + 1;
if (a)
   y++;
else
   y—;
z++;
```



Example of Control Flow Analysis

- How do we tell if the program is buggy?

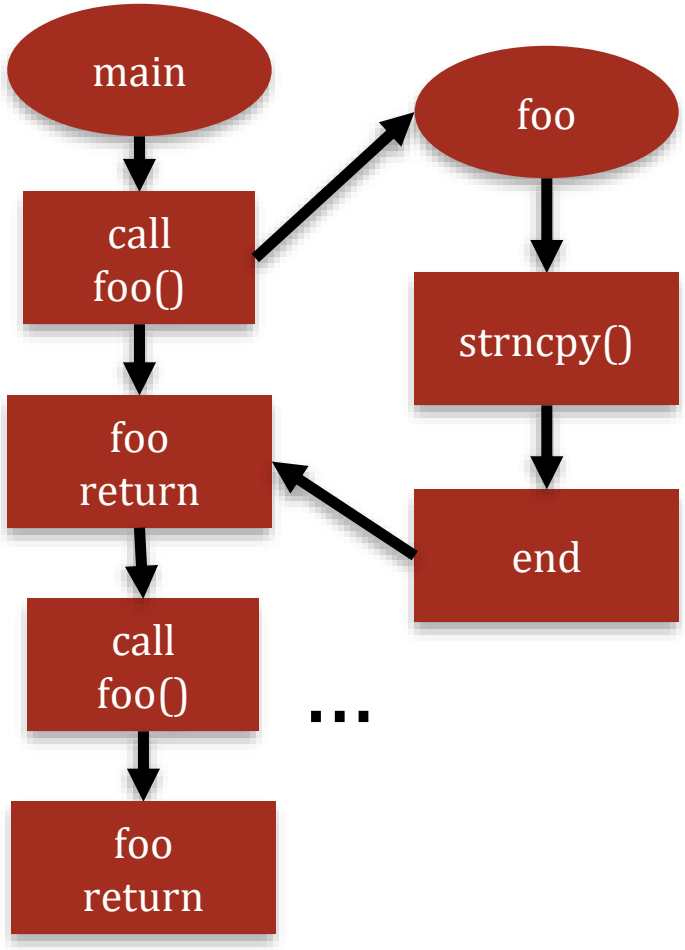
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    cond = foo();
    int size = 0;
    if(cond)
        size = 10;
    else
        size = 100;
    strncpy(buf, argv[1],size);
}
```



Interprocedural CFG

- The basics are easy
  - Call – connect to CFG of callee function
  - Return – create an edge back to the caller function

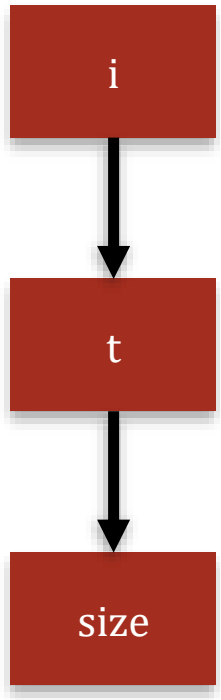
```
#include <string.h>
void foo(int size)
{
    char buf[64];
    strncpy(buf, argv[1], size)
}

int main(int argc, char **argv)
{
    foo(10);  // safe
    foo(100); // buggy
}
```



Data Flow Analysis

- Compute how data flows from one variable to another

```
int main(int argc, char **argv) {
    char buf[64];
    int i;
    scanf("%d", &i);
    int t = i * 2;
    int size = t – 10;
    strncpy(buf, argv[1], size);
}
```



Data/Information Flow Tracking

- Can help with both:
- Secrecy – write programs in which all secret data is only output to authorized subjects
- Integrity – write programs in which there is no way to access adversary-control data that may be used to modify unauthorized data
- These are both achieved by tracking program information flows
  - We will examine information flow in the context of secrecy first

Secrecy Violation of Information Flow

```
SMS sms = readSMS();
String body = sms.body;
String strCopy = body;
socket.send(strCopy);
```

"sms" is secret

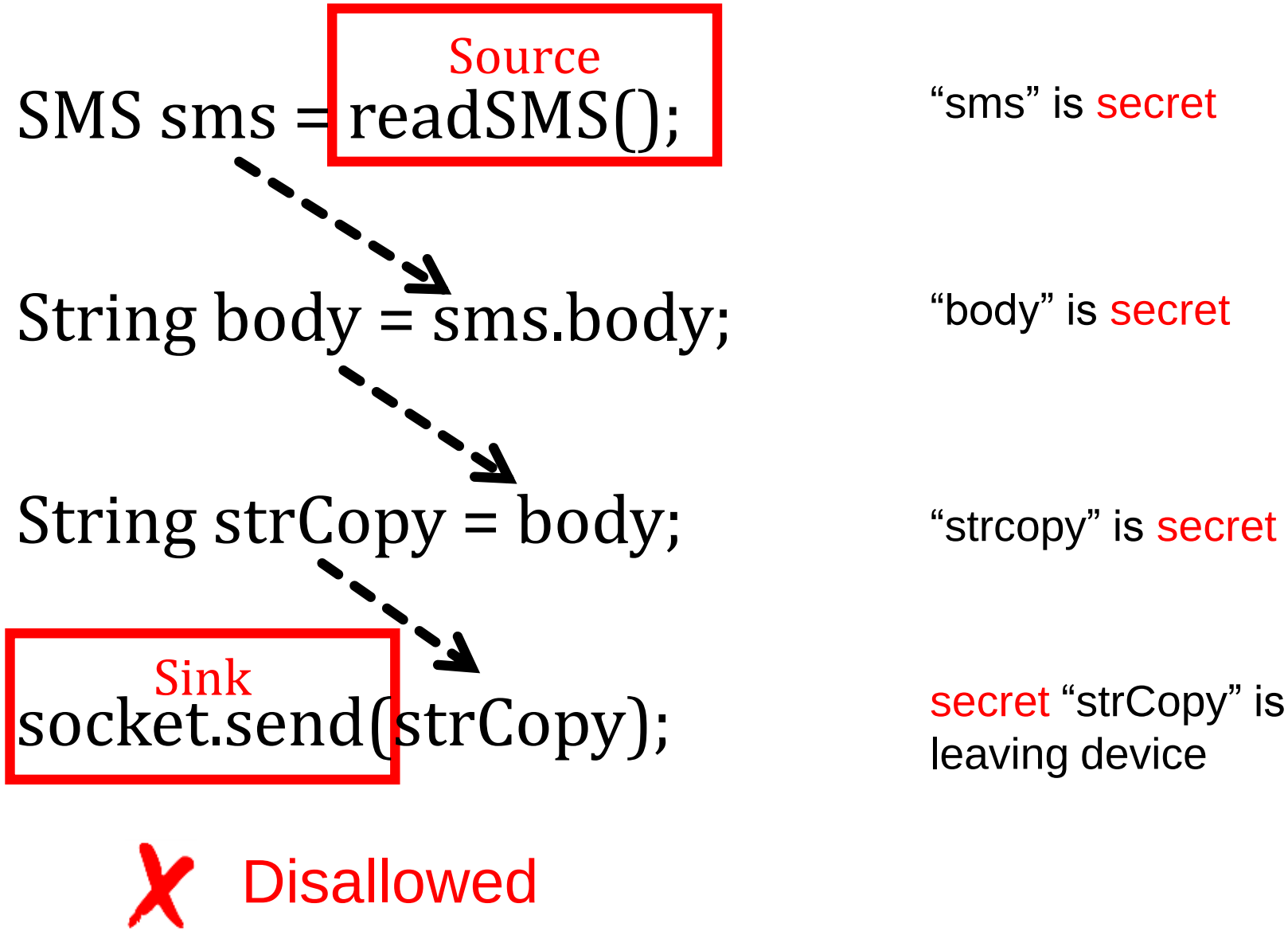
"body" is secret

"strcopy" is secret

secret "strCopy" is leaving device

Disallowed

Secrecy Violation of Information Flow



Integrity Violation of Information Flow

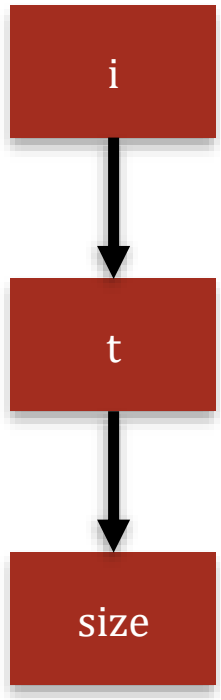
- Compute how data flows from one variable to another

Is this program buggy?

```
int main(int argc, char **argv) {
    char buf[64];
    int i;
    scanf("%d", &i);
    int t = i * 2;
    int size = t - 10;
    strncpy(buf, argv[1], size);
}
```

i is untrusted (source)

strncpy is sink



Integrity Violation of Information Flow

- A more advanced example

```
rc = read( fd, buf, len );  // fd for adversary-controlled file
fd = open( buf, O_RDWR );
```

- What files could be opened (and eventually modified) using this code?

Integrity Violation of Information Flow

- A more advanced example

```
rc = read( fd, buf, len );  // fd for adversary-controlled file
fd = open( buf, O_RDWR );
```

- What files could be opened (and eventually modified) using this code? Any

- How can we prevent such attacks?

Integrity Violation of Information Flow

- Information Flow!

```
rc = read( fd, buf, len ); // fd for adversary-controlled file
fd = open( buf, O_RDWR );
```

- After the first statement – “buf” will be low integrity (adversary-controlled)
- However, the filename used by “open” must be high integrity
  - Information flow error

Integrity Violation of Information Flow

- However, your programs will be chock-full of these

```
rc = read( fd, buf, len ); // fd for adversary-controlled file
fd = open( buf, O_RDWR );
```

- But must perform processing (e.g., store a new object) correctly using that untrusted input
  - How do we do this safely?
  - Checks required: e.g., only files in the /home directory are accessed (declassifier)

Example of Control+Data Flow Analysis

- Can we detect double frees?
  - CFG shows a flow from free(buf2R1) to free(buf2R1)

```
main(int argc, char **argv)
{
     …
     buf1R1 = (char *) malloc(BUFSIZE2);
     buf2R1 = (char *) malloc(BUFSIZE2);
     free(buf1R1);
     free(buf2R1);
     buf1R2 = (char *) malloc(BUFSIZE1);
     strncpy(buf1R2, argv[1], BUFSIZE1-1);
     free(buf2R1);
     free(buf1R2);
}
```

Example of Control+Data Flow Analysis

- Can we detect double frees?
  - CFG shows a flow from free(buf2R1) to free(buf2R1)
- More complex if…
  - Free occurs in a different function
    - Interprocedural CFG (control flow)
  - Free is performed on a different variable
    - Track assignments and aliases (data flow)

Constant Propagation

- Substitute the values of known constants in expressions
- Propagate the values among variables assigned those constants
- Example assignments resulting from propagation to detect problems

Example of Constant Propagation

- What are the constant values below? Is there a buffer overflow?

```
1   char text[] = “Foo                Bar”;
2   char buffer1[4], buffer2[4];
3
4   int i, n = sizeof(text);
5   for(i=0;i<n;++i)
6       buffer2[i] = text[i];
7   printf("Last char of text is: %c", text[n]);
```

Example of Constant Propagation

- Where can they be propagated?

```
1   char text[] = "Foo            Bar";
2   char buffer1[4], buffer2[4];
3
4   int i, n = sizeof(text);
5   for(i=0;i<n;++i)
6       buffer2[i] = text[i];
7   printf("Last char of text is: %c", text[n]);
```

Example of Constant Propagation

- Where are the memory errors?

```
1   char text[] = "Foo            Bar";
2   char buffer1[4], buffer2[4];
3
4   int i, n = 20;
5   for(i=0;i<20;++i)
6       buffer2[i] = text[i];
7   printf("Last char of text is: %c", text[20]);
```

Example of Constant Propagation

- Where are the memory errors?

```
1   char text[] = "Foo                Bar";
2   char buffer1[4], buffer2[4];
3
4   int i, n = 20;
5   for(i=0;i<20;++i)
6       buffer2[i] = text[i];
7   printf("Last char of text is: %c", text[20]);
```

Example of Constant Propagation

- Typically, constant propagation is a start, but need more to detect an error
- For the buffer overflow we need to know that access to buffer2[4-19] and text[20] are memory errors
  - Requires further analysis, e.g., “range analysis” or “symbolic execution”

Type-based Analysis

- Maybe we want to check for certain properties about variables in our program
  - Often by giving variables an extra **label / qualifier**
- Suppose we want to know if a variable's value has been "checked" – such as for input validation
- We can use type-based analysis to do that

Type-based Analysis

- Maybe we want to check for certain properties about variables in our program
- Suppose we want to know if a variable's value has been "checked" – such as for input validation
- We can use type-based analysis to do that

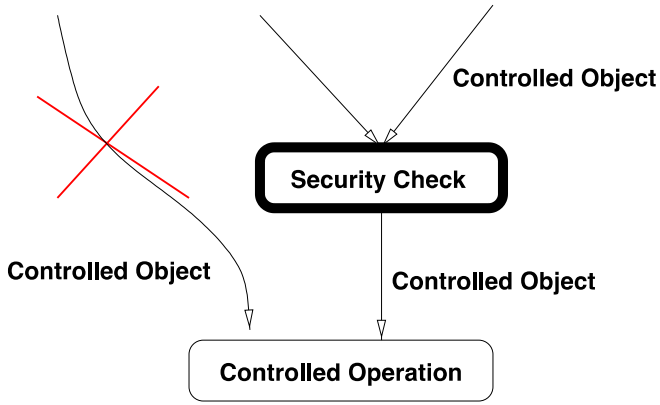
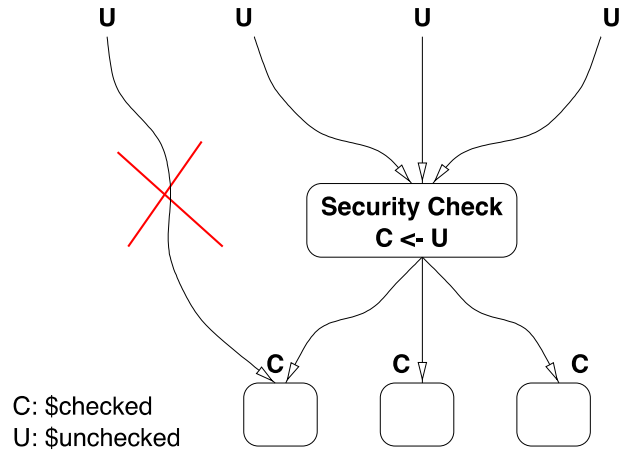


Figure 2: The complete mediation problem.

Type-based Analysis

- Maybe we want to check for certain properties about variables in our program
- Suppose we want to know if a variable's value has been "checked" – such as for input validation
- Using type qualifiers, can extend basic types

```
void func_a(struct file * $checked filp);

void func_b( void )
{
  struct file * $unchecked filp;
  ...
  func_a(filp);
  ...
}
```



Type-based Analysis

- Maybe we want to check for certain properties about variables in our program
- Suppose we want to know if a variable's value has been "checked" – such as for input validation
- To find missing mediation (e.g., input validation)
  - Initialize untrusted inputs to "unchecked"
  - Initialize security-sensitive operation to use "checked"
  - Identify mediation (create "checked" version)
  - Detect type error – from "unchecked" to "checked"

Question

- What do we need to track to discover the vulnerability in project 3?
  - Control flow? Data flow? Constant? Type?

```
void test(char* input)
{
    char test[17] = "abc";
    strcpy(test, input);

    printf("You have input: %s\n", test);
}
void main(int argc, char** args)
{
    if(argc > 1)
    {
        int uid = getuid();
        // the file needs to be generated at a location where normal users cannot touch
        sprintf(filename, "uid_%d_crack", uid);
        printf("file name: %s\n", filename);
        test(args[1]);
    }
    else
    {
        printf("Please provide at least one input\n");
    }
}
```

Questions