CMPSC 447
Static Analysis

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Our Goal

• One option is to develop automated techniques to detect vulnerabilities before they can be exploited
  ‣ Your program may have flaws that may lead to a vulnerabilities
  ‣ How to find them?
Dynamic Analysis Limits

• Major advantage
  ‣ When we produce a crash, it is a real crash

• Major limitation
  ‣ We cannot find all vulnerabilities in a program with dynamic testing in most cases

• Why not?
Dynamic Analysis Limits

• Major advantage
  ‣ When we produce a crash, it is a real crash

• Major limitation
  ‣ We cannot find all vulnerabilities in a program with dynamic testing in most cases

• Why not?
  ‣ Cannot run all possible inputs in most cases
Goal

• Can we build a technique that identifies *all* vulnerabilities?
Goal

• Can we build a technique that identifies *all* flaws?
  ‣ Turns out that we can: static analysis
  ‣ Over-approximate all possible executions of a program, so any flaw that can happen will be found
    • And some flaws that are not really possible (false positives)
  ‣ But, 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 sets of 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
  ‣ Stitch them together to cover all paths

• Runtime testing is inherently incomplete, but static analysis can cover all paths
• Consider the following code

```c
int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf2R2;
    char *buf3R2;

    buf1R1 = (char *) malloc(BUFSIZER1);
    buf2R1 = (char *) malloc(BUFSIZER1);

    free(buf2R1);

    buf2R2 = (char *) malloc(BUFSIZER2);
    buf3R2 = (char *) malloc(BUFSIZER2);

    strncpy(buf2R1, argv[1], BUFSIZER1-1);
    free(argv[1], BUFSIZER1-1);
    free(buf1R1);
    free(buf2R2);
    free(buf3R2);
}
```
Static Analysis

• Can we find a use-after-free flaw?

```c
int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf2R2;
    char *buf3R2;

    buf1R1 = (char *) malloc(BUFSIZER1);
    buf2R1 = (char *) malloc(BUFSIZER1);

    free(buf2R1);

    buf2R2 = (char *) malloc(BUFSIZER2);
    buf3R2 = (char *) malloc(BUFSIZER2);

    strncpy(buf2R1, argv[1], BUFSIZER1-1);
    free(argv[1], BUFSIZER1-1);
    free(buf1R1);
    free(buf2R2);
    free(buf3R2);
}
```
Static Analysis

• Various properties of programs can be tracked
  ‣ Control flow
  ‣ Constants
  ‣ Types
  ‣ Values (sets of values)
  ‣ Data flow

• Which ones will expose which vulnerabilities accurately (and 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 leads to a use-after-free for a pointer
  ‣ That may be run by the program
    • Overapproximates executions
    • For just the part of the program of interest
  ‣ How do we do this?
Intraprocedural CFG

• Statements
  ‣ Nodes
  ‣ One successor and one predecessor

• Basic Blocks
  ‣ Multiple successors (multiple predecessors)

• Unique Enter and Exit
  ‣ All start nodes are successors of enter
  ‣ All return nodes are predecessors of exit
Control Flow for Example

• What is this example’s control flow

```c
int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf2R2;
    char *buf3R2;

    buf1R1 = (char *) malloc(BUFSIZER1);
    buf2R1 = (char *) malloc(BUFSIZER1);

    free(buf2R1);

    buf2R2 = (char *) malloc(BUFSIZER2);
    buf3R2 = (char *) malloc(BUFSIZER2);

    strncpy(buf2R1, argv[1], BUFSIZER1-1);
    free(buf1R1);
    free(buf2R2);
    free(buf3R2);
}
```
Control Flow for Example

• Ah ha! A “use” after a “free”

```c
int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf2R2;
    char *buf3R2;

    buf1R1 = (char *) malloc(BUFSIZER1);
    buf2R1 = (char *) malloc(BUFSIZER1);
    free(buf2R1);
    buf2R2 = (char *) malloc(BUFSIZER2);
    buf3R2 = (char *) malloc(BUFSIZER2);
    strncpy(buf2R1, argv[1], BUFSIZER1-1);
    free(buf1R1);
    free(buf2R2);
    free(buf3R2);
}
```
Control Flow for Example

• Happens to refer to the same pointer

```c
int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf2R2;
    char *buf3R2;

    buf1R1 = (char *) malloc(BUFSIZER1);
    buf2R1 = (char *) malloc(BUFSIZER1);

    free(buf2R1);

    buf2R2 = (char *) malloc(BUFSIZER2);
    buf3R2 = (char *) malloc(BUFSIZER2);

    strncpy(buf2R1, argv[1], BUFSIZER1-1);
    free(buf1R1);
    free(buf2R2);
    free(buf3R2);
}
```
Control Flow for Example

- Would be a false positive otherwise

```c
int main(int argc, char **argv) {
char *buf1R1;
char *buf2R1;
char *buf2R2;
char *buf3R2;

buf1R1 = (char *) malloc(BUFSIZER1);
buf2R1 = (char *) malloc(BUFSIZER1);

free(buf2R1);

buf2R2 = (char *) malloc(BUFSIZER2);
buf3R2 = (char *) malloc(BUFSIZER2);

strncpy(buf2R2, argv[1], BUFSIZER1-1);
free(buf1R1);
free(buf2R2);
free(buf3R2);
}
```
• Reason about possible values (concrete)

```c
int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf2R2;
    char *buf3R2;

    buf1R1 = (char *) malloc(BUFSIZE1);
    buf2R1 = (char *) malloc(BUFSIZE1);

    free(buf1R1);

    buf2R1 = (char *) malloc(BUFSIZE1);
    buf2R2 = (char *) malloc(BUFSIZE2);
    buf3R2 = (char *) malloc(BUFSIZE2);

    strncpy(buf2R1, argv[1], BUFSIZE1-1);
    free(buf2R1);
    free(buf2R2);
    free(buf3R2);
}
```
Control Flow Analysis

• Compute Control Flow

• One function at a time – “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

From Last Time: BB and CFG

- 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 \( E \), \( v_1 \) (BB1) \( \rightarrow \) \( v_2 \) (BB2) if BB2 can immediately follow BB1 in some execution sequence

\[
\begin{align*}
x &= y+1; \\
\text{if (c)} & \quad x++; \\
\text{else} & \quad x--; \\
y &= z + 1; \\
\text{if (a)} & \quad y++; \\
\text{else} & \quad y--; \\
z &= z + 1;
\end{align*}
\]
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
Detect Buffer Overflow

• What are the constant values below?

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]);`
Detect Buffer Overflow

• 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]);`
Detect Buffer Overflow

• 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]);`
Detect Buffer Overflow

• Where are the memory errors?

```c
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]);
```
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
Abstract Interpretation

• Descriptors represent the sign of a value
  ‣ Positive, negative, zero, unknown

• For an expression, $c = a \times b$
  ‣ If $a$ has a descriptor $pos$
  ‣ And $b$ has a descriptor $neg$

• What is the descriptor for $c$ after that instruction?
• How might this help?
Abstract Interpretation

• E.g., integer overflows
• Use unknown for signed ints
• And “<constant” for signed after (signed < constant)
• “Cast_unsigned” creates a positive from <constant
• Could we detect a problem here?

```c
if (signed < constant)
    strlcpy(dst, src, (cast_unsigned)signed);
```
Type-based Analysis

• Maybe we want to check for certain properties about variables in our program

• Can use type information associated with variables to perform such checks
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
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
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

```c
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”
Type-based Analysis

- Vulnerability in the code to the right
  - Can you see it?
Type-based Analysis

• Vulnerability in the code to the right
  ‣ fd is unchecked as is filp initially in sys_fnctl
  ‣ However, filp would be reassigned to a checked variable after security_op

• So what’s the problem?

```c
/* from fs/fcntl.c */
long sys_fcntl(unsigned int fd,
              unsigned int cmd
              unsigned long arg)
{
  struct file * filp;
  ...
  filp = fget(fd);
  ...
  err = security_ops->file_ops
         ->fcntl(filp, cmd, arg);
  ...
  err = do_fcntl(fd, cmd, arg, filp);
  ...
}
static long
do_fcntl(unsigned int fd,
         unsigned int cmd,
         unsigned long arg,
         struct file * filp) {
  ...
  switch(cmd){
    ...
    case F_SETLK:
      err = fcntl_setlk(fd, ...);
      ...
  }
  ...
}
/* from fs/fcntl.c */
sys_fnctl(fd, ...
       { struct file * filp;
         ...
         filp = fget(fd);
         /* operate on filp */
         ...
}
Type-based Analysis

- Vulnerability in the code to the right
  - $fd$ and $filp$ are unchecked initially
  - $filp$ is checked in `sys_fnctl`
  - However, $filp$ is reassigned from an unchecked $fd$ variable in `fnctl_getlk/setlk`

- $fd$, not the checked $filp$ is passed to `do_fcntl` and to `fnctl_getlk/setlk`

```c
/* from fs/fcntl.c */
long sys_fcntl(unsigned int fd,
              unsigned int cmd,
              unsigned long arg)
{
  struct file * filp;
  ...
  filp = fget(fd);
  ...
  err = security_ops->file_ops
    ->fcntl(fd, cmd, arg);
  ...
  err = do_fcntl(fd, cmd, arg, filp);
  ...
}
```

```c
static long
do_fcntl(unsigned int fd,
          unsigned int cmd,
          unsigned long arg,
          struct file * filp) {
  ...
  switch(cmd){
    ...
    case F_SETLK:
      err = fnctl_setlk(fd, ...);
    ...
  }
  ...
}
```

```c
/* from fs/locksys.c */
fnctl_getlk(fd, ...) {
  struct file * filp;
  ...
  filp = fget(fd);
  /* operate on filp */
  ...
}
```
• Static analysis evaluates all the ways that a program may execute in one pass
  ‣ Can be “sound” (no false negatives – find all flaws)
  ‣ But, then will likely produce some false positives

• Examined some building blocks of static analysis and how they could be used
  ‣ Constant propagation, control flow, type analysis

• There is much more to the application of static analysis to security problems – a key for software security