Vulnerability Detection - Static Analysis

Holistic Software Security

Aravind Machiry

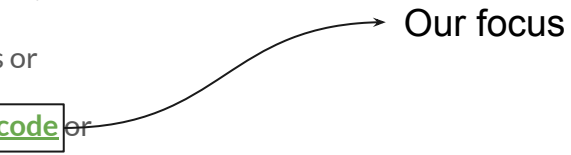
Credits: Prof. Mayur Naik, UPenn

What is it?

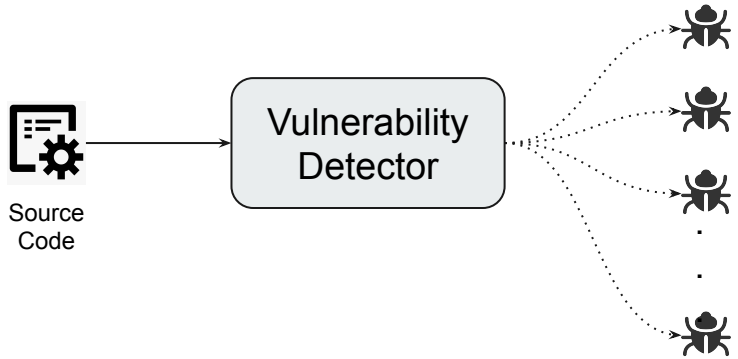
- Finding vulnerabilities in a given piece of software:
  - Software could be:
    - Binaries or
    - Source code or
    - Both.

What is it?

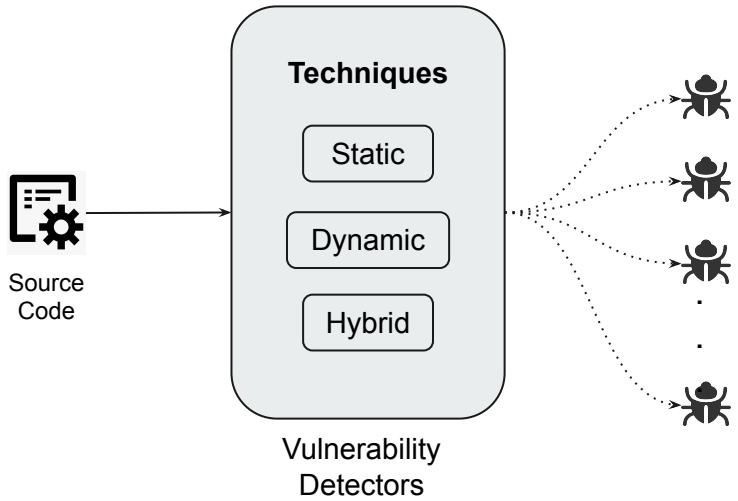
- Finding vulnerabilities in a given piece of software:
  - Software could be:
    - Binaries or
    - Source code or
    - Both.



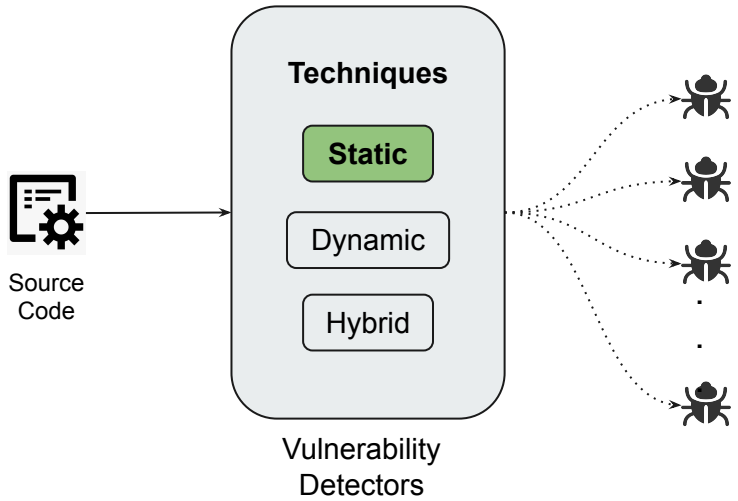
Overview



Overview


Techniques
Static
Dynamic
Hybrid
Source Code
Vulnerability Detectors

Overview


Techniques
Static
Dynamic
Hybrid
Source Code
Vulnerability Detectors

Static

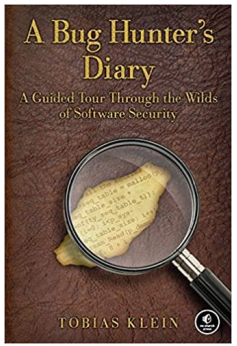
- Static w.r.t to the software being analysed:
  - We **do not run** (or dynamically execute) the program.
- Example:
  - `grep -r "sscanf[^)]*,[^)]*%s"`
  - To find: CWE-120 - Buffer Copy without Checking Size of Input ('Classic Buffer Overflow').

Static

- `grep -r "sscanf[^)]*,[^)]*%s"`

```
static char cs;
...
int ret = sscanf(buf, "%s", &cs);
if (ret != 1) {
   accdet_error("..");
   return -1;
}
...
```

CVE-2016-8472: In MediaTek Kernel Driver



Most successful technique

Static

- `grep -r "sscanf[^)]*,[^)]*%s" -> CVE-2016-8472`
  - Along with **2,300** other matches which **are not vulnerabilities (False positives)**.

```
...
char *ptr = “CMD 12”;
char buf[64]
...
sscanf(ptr, "%s", buf);
```

Maximum size could be 6 (less than 64 -> size of buf)

It becomes worse on complex codebases!

	CppCheck	flawfinder	RATS
Qualcomm	18	4,365	693
Samsung	22	8,173	2,244
Huawei	34	18,132	2,301
MediaTek	168	14,230	3,730
Total	**242**	**44,990**	**8,968**

Static

- How does a human find vulnerabilities?

```
void overflow() {
  char *out;
  int in = get_int();
  if (in <= 0) { return; }
  out = malloc(in*sizeof(char*));
  for (i = 0; i < in; i++)
    out[i] = get_string();
}
```

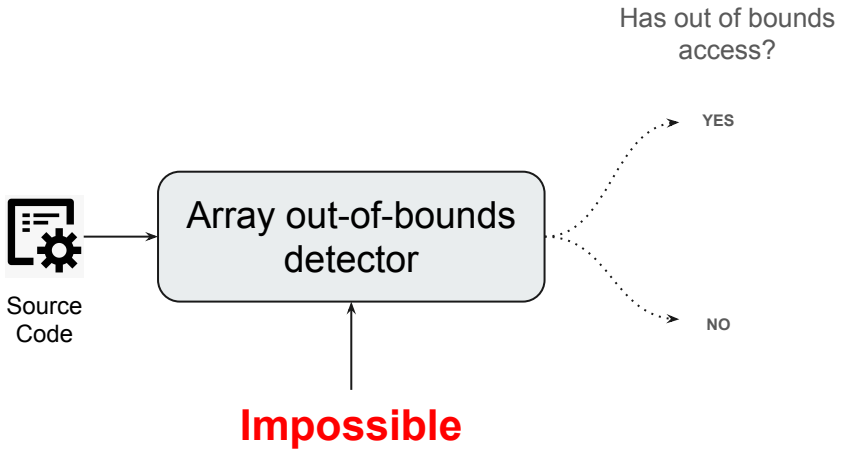
1073741824

0

Static

- How does a human find vulnerabilities?
  - Understands the program and tries to find if any vulnerable conditions are possible.
  - We need a way to analyze the given program or software:
    - Program Analysis -> **Static Program Analysis** or **Static Analysis**

But, computing program properties is undecidable!



But, computing program properties is undecidable!

```
void foo() {
  int a[2];
  M(X);
  a[3] = 0;
}
```

- Halting Problem: Impossible to say whether a program terminates.
- Proof by contradiction:
  - Yes -> Execution reaches a[3] i.e., program M(X) terminates.
  - No -> Execution does not reach a[3] i.e., program M(X) does not terminate.
- Contradiction: We can say if a program terminates.

Static analysis design choices for vulnerability detection

Impossible

True Result	Sound	Complete	Neither sound nor complete	Sound and Complete
Bug	Bug	May or May not be a bug.	May or May not be a bug.	Bug
Not a bug	May or May not be a bug.	Not a bug.	May or May not be a bug.	Not a bug
	false positives No false negatives	No false positives false negatives	false positives false negatives	No false positives No false negatives

Precision and Recall

		Analysis Outcome	
		Accept	Reject
Program's Ground Truth	Good	True Negative	False Positive
	Bad	False Negative	True Positive

$$precision = \frac{\#\ \text{True Positives}}{\#\ \text{Rejected}}$$

$$recall = \frac{\#\ \text{True Positives}}{\#\ \text{Bad}}$$

Static analysis design choices for vulnerability detection

	Recall=1	Precision=1		
True Result	**Sound**	**Complete**	**Neither sound nor complete**	**Sound and Complete**
Bug	Bug	May or May not be a bug.	May or May not be a bug.	Bug
Not a bug	May or May not be a bug.	Not a bug.	May or May not be a bug.	Not a bug
	false positives No false negatives	No false positives false negatives	false positives false negatives	No false positives No false negatives

Sound Static Analysis

- **Used to be the** popular choice. Why?
  - Guarantees that all bugs will be found.
    - Over Approximation.
    - Caveat: False positives.
  - If a sound static analysis says, there are no bugs*, then we can be sure that the program does not have bugs.

* of specific type.

Sound Static Analysis

```
void foo(unsigned i) {
  int a[2];
 if (i < 2) a[i] = 0; //p3
 else a[i] = 1; //p4
}

int main() {
  unsigned i, j;
  scanf(“%u %u”, &i, &j);
  if (i < 2) foo(i); //p1
  foo(j); //p2
  return 0;
}
```

Consider the following out-of-bounds detectors with the following warnings at corresponding lines:

- SA1: P1, P2, P3, P4
- SA2: P3 and P4
- SA3: P4
- SA4: P4 only when called from P2

Are these analyses sound?

Sound Static Analysis

```
void foo(unsigned i) {
  int a[2];
 if (i < 2) a[i] = 0; //p3
 else a[i] = 1; //p4
}

int main() {
  unsigned i, j;
  scanf(“%u %u”, &i, &j);
  if (i < 2) foo(i); //p1
  foo(j); //p2
  return 0;
}
```

Consider the following out-of-bounds detectors with the following warnings at corresponding lines:

- SA1: P1, P2, P3, P4
- SA2: P3 and P4
- SA3: P4
- SA4: P4 only when called from P2

Are these analyses sound?

What about precision?

Designing a Sound Static Analysis

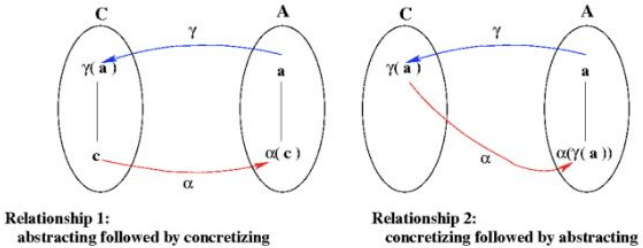
- Guaranteed Termination: Should finish in reasonable time.
- **Over Approximate** program behavior.

Abstract Interpretation

- Interpret the program over abstract states.
- Abstract semantics:
  - How to interpret operations over abstract values.
- Guaranteed Termination (Kleene fixed-point theorem):
  - Galois Connection.
  - Monotonic Transfer functions:
    - The state computed at a program point should never decrease.

Abstract Interpretation

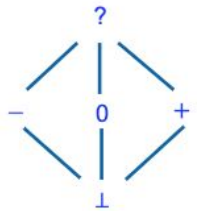
- Galois Connection:
  - Abstraction function (α) -> Maps a set of concrete values to abstract value.
  - Concretization function (γ) -> Maps an abstract value to set of concrete values.
  - **1. α(c) ≤ a ⇐⇒ c ≤ γ(a)**
  - **2. α(γ(a)) ≤ a**



Sign Abstract Domain

To handle properties related to integers.

Abstract Values: {-, 0, +, ⊥, ?}



$$\alpha(S) = \begin{cases} 0 & \text{if all elements of S are 0} \\ + & \text{if all elements of S are positive} \\ - & \text{if all elements of S are negative} \\ ? & \text{otherwise} \end{cases}$$

$$\gamma(S) = \begin{cases} \{0\} & \text{if } S = 0 \\ \{\text{pos int}\} & \text{if } S = + \\ \{\text{neg int}\} & \text{if } S = - \\ \{0 \text{ pos neg}\} & \text{if } S = ? \end{cases}$$

ADD	-	0	+	?
-	-	-	?	?
0	-	0	+	?
+	?	+	+	?
?	?	?	?	?

MULT	-	0	+	?
-	+	0	-	?
0	0	0	0	0
+	-	0	+	?
?	?	0	?	?

Divide by Zero Detector

- We do not care about absolute values of integers.
- We **just need to know if a number can be 0 or not**.
- Sign abstract domain provides a decent choice.
- Possible values for numbers: {-, 0, +, ⊥, ?}

```
void main() {
    ...
    if (x > 0) {
        ...1/x.. // x:+
    }
    ...2/x.. // x: ?
}
```

numRequests: ?

```
int averageResponseTime(int totalTime, int numRequests) {
  return totalTime / numRequests;
}
```

CVE-2019-14498
A divide-by-zero error exists in VLC media player that can be exploited by a crafted audio file

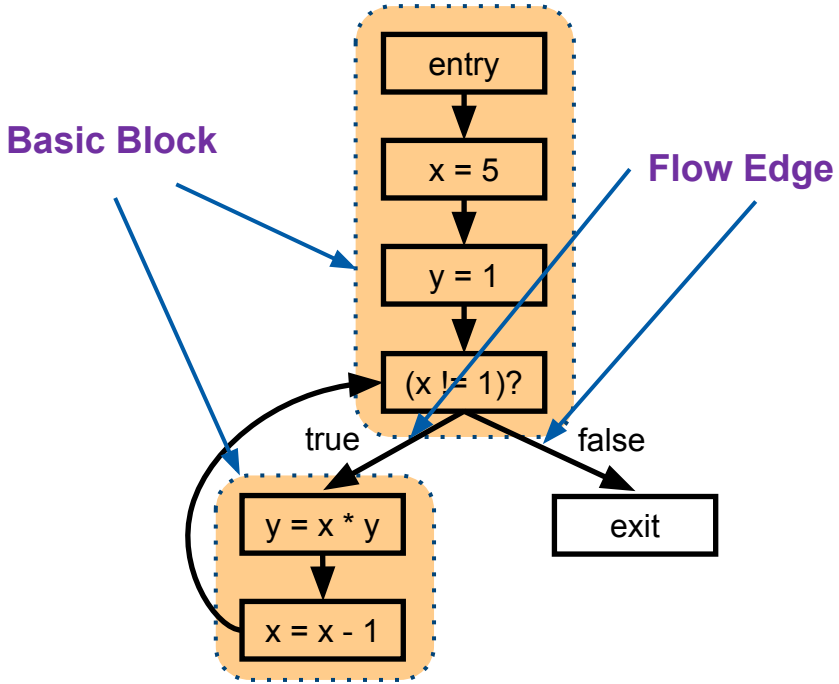
Data flow analysis

- Most vulnerabilities need reasoning of the flow of data through the program.
    - E.g., user input used as an index into an array => User data flows into index of an array.
- Reasoning about flow of data in programs.
- Different kinds of data: constants, expressions, taint, etc.

Data flow concepts

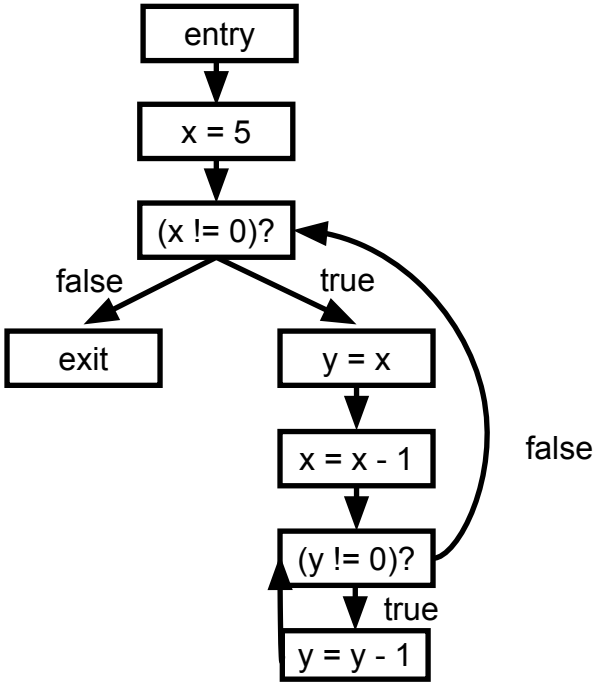
- Control flow graph (CFG):
  - Represents possible control flows with in the function.
  - Graph of basic blocks.
  - **Basic block:** Sequence of instructions always executed in the order.
  - **Edges** -> Flow of control.

Control flow graph (CFG)




```
x = 5;
y = 1;
while (x != 1) {
  y = x * y;
  x = x - 1
}
```

Control flow graph (CFG)




```
x = 5;
while (x != 0) {
  y = x;
  x = x - 1;
  while (y != 0) {
    y = y - 1
  }
}
```

Classic Dataflow Analyses -> Primarily used in compiler optimization

Reaching Definitions Analysis

- Find uninitialized variable uses

Very Busy Expressions Analysis

- Reduce code size

Available Expressions Analysis

- Avoid recomputing expressions

Live Variables Analysis

- Allocate registers efficiently

Security related Dataflow Analyses

Interval Analysis

- Check memory safety
  (integer overflows, buffer overruns, …)

Type-State Analysis

- Temporal safety properties
  (APIs of protocols, libraries, …)

Taint Analysis

- Check information flow
  (Sensitive data leak, code injection, …)

Concurrency Analysis

- Concurrency safety properties
  (dataraces, deadlocks, …)

Reaching Definition Analysis

Determine, for each program point, which assignments (definitions) have been made and not overwritten, when execution reaches that point along some path.