Course Introduction

Week 1

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Administration

- Course convenor and lecturer: A/Prof. Yulei Sui
- Email: cs6131@cse.unsw.edu.au
- Course webpage: `https://webcms3.cse.unsw.edu.au/COMP6131/24T2`
- Course forums: `https://edstem.org/au/courses/16128/discussion`
- Important messages will be announced on the course homepage, and urgent messages will also be sent to you via email.
- Course keywords: Source Code Analysis, Security vulnerabilities, Verification, Abstract Interpretation

Course Aim

In this course, you will learn to create automated **code analysis and verification tools** using a **modern compiler** and an **open-source** static analysis framework, to perform **code comprehension**, **vulnerability detection** and code **verification** in real-world software systems.

Learning Outcomes

- Practice **system programming skills** to develop **code analysis and verification techniques** to address code security and reliability problems.
  - **High-quality coding**: Commit to writing high-quality, error-free, and high-performance code, especially within the context of large-scale codebases.
  - **Compiler basics**: Gain insights into compilation, code representation, low-level instructions, code debugging and profiling.
  - **Vulnerability Comprehension**: Understand common vulnerabilities, such as tainted information flow, buffer overflows, and assertion errors.
  - **Open-source static analysis framework**: learn to build practical tools on top of open-source frameworks like SVF.
  - **Formal Verification**: Understand formal methods and techniques for verifying code correctness using mathematical and logical reasoning tools.

Learning Outcomes

- Practice **system programming skills** to develop **code analysis and verification techniques** to address code security and reliability problems.
  - **High-quality coding**: Commit to writing high-quality, error-free, and high-performance code, especially within the context of large-scale codebases.
  - **Compiler basics**: Gain insights into compilation, code representation, low-level instructions, code debugging and profiling.
  - **Vulnerability Comprehension**: Understand common vulnerabilities, such as tainted information flow, buffer overflows, and assertion errors.
  - **Open-source static analysis framework**: learn to build practical tools on top of open-source frameworks like SVF.
  - **Formal Verification**: Understand formal methods and techniques for verifying code correctness using mathematical and logical reasoning tools.
- Career and job roles
  - Software Engineer; Security Analyst/Engineer; Compiler Engineer; Formal Methods Engineer; Software Reliability Engineer; Embedded Systems Developer; Research Scientist (in Academia or Industry);

Teaching Rationale and Strategy

Three major components of this course: (1) **Lectures**, (2) **Labs** and (3) **Assignments**. This is a project-based course and **no paper examination** is required.

Teaching Rationale and Strategy

Three major components of this course: (1) **Lectures**, (2) **Labs** and (3) **Assignments**. This is a project-based course and **no paper examination** is required.

- **Lectures** (2 hours per week): Foundational theories and techniques of static code analysis and verification aimed at detecting vulnerabilities and verifying the absence of bugs in system code.

Teaching Rationale and Strategy

Three major components of this course: (1) **Lectures**, (2) **Labs** and (3) **Assignments**. This is a project-based course and **no paper examination** is required.

- **Lectures** (2 hours per week): Foundational theories and techniques of static code analysis and verification aimed at detecting vulnerabilities and verifying the absence of bugs in system code.
- **Labs** (2 hours per week): Hands-on experience including as **preparatory activities before each assignment** through completing
  - **Quizzes**
  - **Coding exercises** (small-scale)

Teaching Rationale and Strategy

Three major components of this course: (1) **Lectures**, (2) **Labs** and (3) **Assignments**. This is a project-based course and **no paper examination** is required.

- **Lectures** (2 hours per week): Foundational theories and techniques of static code analysis and verification aimed at detecting vulnerabilities and verifying the absence of bugs in system code.
- **Labs** (2 hours per week): Hands-on experience including as **preparatory activities before each assignment** through completing
  - **Quizzes**
  - **Coding exercises** (small-scale)
- **Three assignments**: Each assignment is built on the previous one to develop code-checking tools capable of:
  - tracking tainted information flows
  - performing symbolic execution
  - conducting abstract interpretation

Teaching Rationale and Strategy

Three major components of this course: (1) **Lectures**, (2) **Labs** and (3) **Assignments**. This is a project-based course and **no paper examination** is required.

- **Lectures** (2 hours per week): Foundational theories and techniques of static code analysis and verification aimed at detecting vulnerabilities and verifying the absence of bugs in system code.
- **Labs** (2 hours per week): Hands-on experience including as **preparatory activities before each assignment** through completing
  - **Quizzes**
  - **Coding exercises** (small-scale)
- What would be the best practice for completing an assignment (e.g., Assignment-1)?
  - **Correct way**: Quiz-1 $\rightarrow$ Lab-Exercise-1 $\rightarrow$ Assignment-1
  - **Incorrect way**: all other orders

Course Schedule

Week	Content	Quiz & Exercise	Assignment
1	**Lecture**: Course Overview and Introduction **Lab**: C++ practices, vulnerability assessment and compiler IR	-	-
2	**Lecture**: Control and Data Flows **Lab**: Code graphs, SVF, constraints solving	Lab Exercise 1 (10%)	-
3	**Lecture**: Pointer Aliasing and Taint Tracking **Lab**: Tainted information flow tracking	-	Assignment 1 (20%)
4	**Lecture**: Code Verification Basis **Lab**: Verification concepts, predicate logic	-	-
5	**Lecture**: Automated Theorem Proving **Lab**: Manual assertion-based verification using Z3	Lab Exercise 2 (10%)	-
6	**Flexibility Week**	-	-
7	**Lecture**: Code Verification using Symbolic Execution **Lab**: Automated code assertion verification using Z3	-	Assignment 2 (25%)
8	**Lecture**: Abstract Interpretation Foundations **Lab**: Basic concepts and examples	Lab Exercise 3 (10%)	-
9	**Lecture**: Code Verification using Abstract Interpretation **Lab**: Manual assertion-based verification using Z3	-	-
10	**Lecture**: Buffer Overflow Detection using Abstract Interpretation **Lab**: Implementation and testing	-	Assignment 3 (25%)

Marking and Plagiarism

Marking and late submission:

- Please refer to rubrics for each assessment before you start at Canvas
- Late submission is strongly discouraged. Late submission of assignments will incur the following penalties: 10% of the total possible mark for that piece of assignment for each day past the deadline
- For example, if an assessment receives a mark of 70% out of 100% and is one day late, it will receive a mark of 60% out of 100%

Plagiarism:

- UNSW will adopt a uniform set of penalties for UNSW courses
- Please refer the course outline
- `https://webcms3.cse.unsw.edu.au/COMP6131/24T2/outline`

Course Materials and Resources

No single textbook covers all the course content. Recommended references and an abundance of online materials are available below:

- Static Value-Flow Analysis Framework for Source Code
  - https://github.com/SVF-tools/Teaching-Software-Verification
  - https://github.com/SVF-tools/SVF
- Compilers: Principles, Techniques, and Tools Hardcover, https://www.amazon.com.au/Compilers-Alfred-V-Aho/dp/0321486811
- LLVM Compiler https://llvm.org/
- Symbolic Execution https://en.wikipedia.org/wiki/Symbolic_execution
- Abstract Interpretation https://en.wikipedia.org/wiki/Abstract_interpretation
- Z3 Theorem Prover
  - https://github.com/Z3Prover/z3
  - https://theory.stanford.edu/~nikolaj/programmingz3.html

Introduction to Software Security Analysis

(Week 1)

Yulei Sui

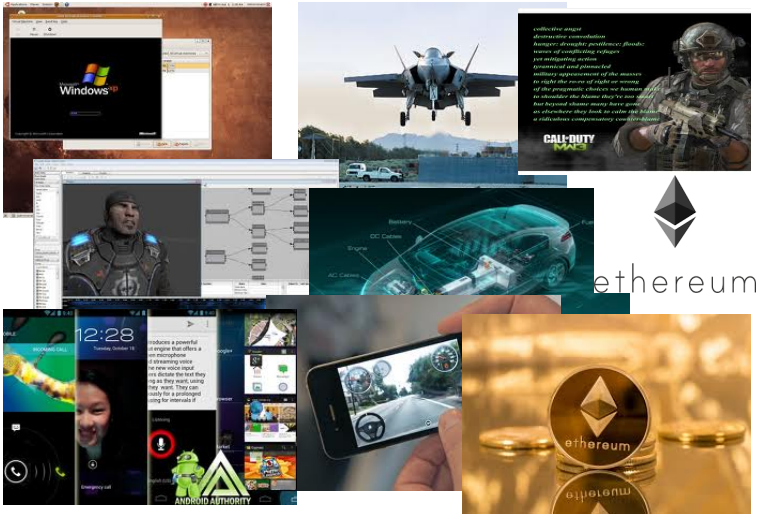
School of Computer Science and Engineering

University of New South Wales, Australia

Outline

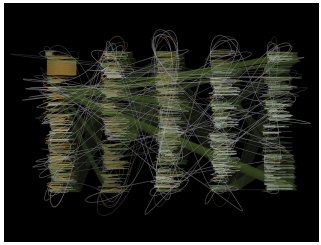
- Background and Introduction to Software Analysis and Verification
- Software Vulnerability Assessment
- Compiler and Code Graphs

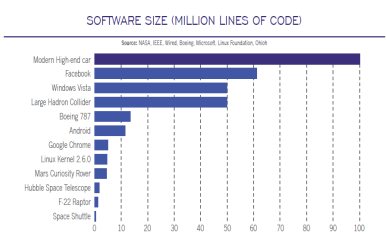
Software Is Everywhere

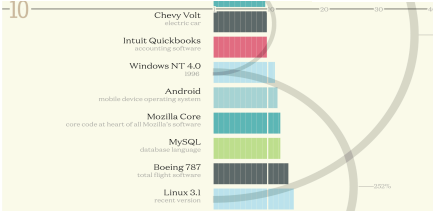


Modern System Software

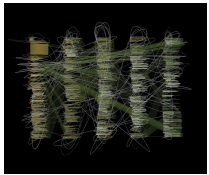
– Extremely Large and Complex

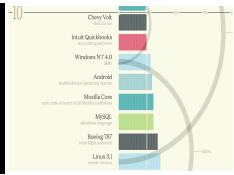






Software Becomes More Buggy

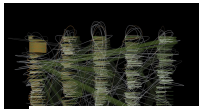




More Complex!

More Buggy!

Software Becomes More Buggy



More Complex!

Vulnerabilities (security defects)

Design apps to run in cloud

The risks

Quality issue: many more "underwater" than those reported "above the water"

The National Vulnerability Database (DHS/US-CERT)

- Lists >47,000 documented vulnerabilities

Undiscovered/unreported (0-day) vulnerabilities are huge

- 20X[1] multiplier
- 47,000 x 20 = estimated **940,000 vulnerabilities** replicated in many products

Greater than 80% of attacks happen at the application layer

Public vulnerabilities are tip of the iceberg !

More Buggy!

https://www.slideshare.net/innotech_conference/hp-cloud-security-inno-tech-20140501

Let us take a look at some real-world vulnerability examples ...

Memory Leak

- A dynamically allocated object is not freed along some execution path of a program
- A major concern of long running server applications due to gradual loss of available memory.

```
1  /* CVE-2012-0817 allows remote attackers to cause a denial of service through adversarial connection requests.*/
2  /* Samba --libads/ldap.c:ads_leave_realm */.
3
4  host = memAlloc(hostname);
5  ...
6  if (...)   {...; return ADS_ERROR_SYSTEM(ENOENT);} // The programmer forgot to release host on error.
7
```

```
1  /* A memory leak in Php-5.5.11 */
2  for (...) {
3      char* buf = readBuffer();
4      if (condition)
5          printf (buf);
6      else
7          continue;   // buf is leaked in else branch
8      freeBuf(buf);
9  }
```

Buffer Overflow

- Attempt to put more data in a buffer than it can hold.
- Program crashes, undefined behavior or zero-day exploit[1].

```
1  /* A simplified example from "Young and Mchugh, IEEE S&P 1987", exploited by attackers to bypass verfication*/
2
3    void verfiyPassword(){
4      char buff[15]; int pass = 0;
5       printf ("\n Enter the password : \n");
6      gets(buff);
7
8       if (strcmp(buff, "thegeekstuff")){     // return non-zero if the two strings do not match
9            printf ("\n Wrong Password \n");
10     }
11     else{                                   // return zero if two strings matched or a buffer overrun
12           printf ("\n Correct Password \n");
13          pass = 1;
14     }
15     if (pass)
16           printf ("\n Root privileges given to the user \n");
17   }
18
```

Uninitialized Variable

- Stack variables in C and C++ are not initialized by default.
- Undefined behavior or denial of service via memory corruption

```
1  /* An uninitialized variable vulnerability simplified from gnuplot (CVE-2017-9670) */
2
3  void load(){
4          switch ( ctl ) {
5                  case -1:
6                          xN = 0; yN = 0;
7                          break;
8                  case 0:
9                          xN = i; yN = -i;
10                         break;
11                 case 1:
12                         xN = i + NEXT_SZ; yN = i - NEXT_SZ;
13                         break;
14                 default:
15                         xN = -1; xN = -1; // xN is accidentally set twice while yN is uninitialized
16                         break;
17         }
18                 plot(xN, yN);
19 }
20
21
```

Use-After-Free

- Attempt to access memory after it has been freed.
- Program crashes, undefined behavior or zero-day exploit.

```
1  /*  CVE-2015-6125 and CVE-2018-12377 with similar heap use after free patterns*/
2
3  char* msg = memAlloc(...);
4  ...
5  if (err) {
6          abrt = 1;
7           ....
8          free(msg); // the memory is released when an error occurs at server
9  }
10 ...
11 if (abrt) {
12          ...
13          logError("operation aborted before commit", msg); // try to access released heap variable,
14                                                            // causing either crash or writing confidential data
15 }
```

Code Review by Developers

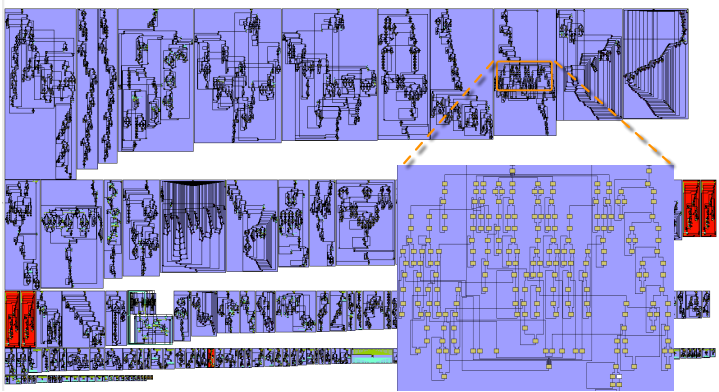
However ...

A large project (e.g., consists of millions of lines of code) is almost impossible to be manually checked by human :

-- intractable due to potentially unbounded number of paths that must be analyze
-- undecidable in the presence of dynamically allocated memory and recursive data structures

How about real-world large programs?

Whole-Program CFG of 300.twolf (20.5K lines of code)



#functions: 194

#pointers: 20773 #loads/stores: 8657 Costly to reason about flow of values on CFGs!

Source Code Analysis and Verification

Automatically analyzing and assuring the behavior of computer programs regarding a property such as correctness, robustness, safety and liveness.

- **Code Analysis** (Week 1-4, Assignment 1)
  - Aim to ***find existence of bugs***, e.g., If there exist a path, a bug can/may be triggered
- **Code Verification** (Week 5-10, Assignments 2 and 3)
  - Aim to ***prove absence of bugs***, e.g., For all paths, user specification should be satisfied and no bug should be triggered.

Software Verification

- Software Verification is useful for proving the correctness, safety and security of a program, and a key aspect of testing can execute as expected.
  - ”Have we made what we were trying to make?”
    - Are we building the system right?
    - Does our design meet the user expectations?
    - Does the implementation conform to the specifications?

Why Software Analysis and Verification?

- Better quality in terms of more secure and reliable software
  - Help reduce the chances of system failures and crashes
  - Cut down the number of defects found during the later stages of development
  - Rule out the existence of any backdoor vulnerability to bypass a program's authentication
- Reduce time to market
  - Less time for debugging.
  - Less time for later phase testing and bug fixing
- Consistent with user expectations/specifications
  - Assist the team in developing a software product that conforms to the specified requirements
  - Help get a better understanding of (legacy) parts of a software product

What Types of Analysis/Verification We Have?

Code verification vs design verification

- **Design approach**: analyzing and verifying the design of a software system.
  - Design specs: specification languages for components of a system. For example,
    - Z language for business requirements,
    - Promela for Communicating Sequential Processes
    - B method based on Abstract Machine Notation.
    - Specification Language (VDM-SL)
    - ...

What Types of Analysis/Verification We Have?

Code verification vs design verification

- **Design approach**: analyzing and verifying the design of a software system.
  - Design specs: specification languages for components of a system. For example,
    - Z language for business requirements,
    - Promela for Communicating Sequential Processes
    - B method based on Abstract Machine Notation.
    - Specification Language (VDM-SL)
    - ...
- **Code approach**: verifying correctness of source code (**This subject**)
  - Code specs (e.g., return a sorted list):
    - Assertions and pre/postconditions in Hoare logic (design by contract)
    - Type systems
    - Well-formed comments or annotations
    - ...

How to Perform Analysis and Verification?

Prove or disprove the correctness of your code against the specifications via

- **Dynamic approach** (Checking code behavior during program execution)
  - **Per path verification** which aims to find bugs by exercising one execution path a time based on specific testing inputs
    - **Stress testing**
    - **Model-based testing**
    - **Fuzz testing**
    - . . .

How to Perform Analysis and Verification?

Prove or disprove the correctness of your code against the specifications via

- **Dynamic approach** (Checking code behavior during program execution)
  - **Per path verification** which aims to find bugs by exercising one execution path a time based on specific testing inputs
    - **Stress testing**
    - **Model-based testing**
    - **Fuzz testing**
    - ...
- **Static approach** (inspecting the code before it runs) (**This subject**)
  - **All path verification** which aims to prove that a program satisfies the specification of its behavior by reasoning about all possible program paths
    - **Control- and Data-flow analysis** (computing control- and data-dependence of a program on code graphs to collect reachability properties) - *Assignment 1*

How to Perform Analysis and Verification?

Prove or disprove the correctness of your code against the specifications via

- **Dynamic approach** (Checking code behavior during program execution)
  - **Per path verification** which aims to find bugs by exercising one execution path a time based on specific testing inputs
    - **Stress testing**
    - **Model-based testing**
    - **Fuzz testing**
    - ...
- **Static approach** (inspecting the code before it runs) (**This subject**)
  - **All path verification** which aims to prove that a program satisfies the specification of its behavior by reasoning about all possible program paths
    - **Control- and Data-flow analysis** (computing control- and data-dependence of a program on code graphs to collect reachability properties) - *Assignment 1*
    - **Symbolic execution** (a practical way to use symbolic expressions instead of concrete values to explore the possible program paths) - *Assignment 2*

How to Perform Analysis and Verification?

Prove or disprove the correctness of your code against the specifications via

- **Dynamic approach** (Checking code behavior during program execution)
  - **Per path verification** which aims to find bugs by exercising one execution path a time based on specific testing inputs
    - **Stress testing**
    - **Model-based testing**
    - **Fuzz testing**
    - ...
- **Static approach** (inspecting the code before it runs) (**This subject**)
  - **All path verification** which aims to prove that a program satisfies the specification of its behavior by reasoning about all possible program paths
    - **Control- and Data-flow analysis** (computing control- and data-dependence of a program on code graphs to collect reachability properties) - *Assignment 1*
    - **Symbolic execution** (a practical way to use symbolic expressions instead of concrete values to explore the possible program paths) - *Assignment 2*
    - **Abstract interpretation** (a general theory of sound approximation of a program through program abstractions or abstract values) - *Assignment 3*

Assertions as Specifications in Software Verification

Assertions are program statements used to **test assumptions** made by software designers/programmers/testers. An assertion is a predicate or an expression that **always should evaluate to true** at that point during code execution.

```
assert(expr);
```

unfold →

```
if(expr is true){
    // continue normal execution
}
else{
    __assert_fail();
    // program failure and terminate the program
}
```

Assertions as Specifications in Software Verification (Example)

- Assertions are program statements used to **test assumptions** made by software designers/programmers/testers.
- An assertion is a expression/predicate that **always should evaluate to true** at that point during code execution.

Assertions as Specifications in Software Verification (Example)

- Assertions are program statements used to **test assumptions** made by software designers/programmers/testers.
- An assertion is a expression/predicate that **always should evaluate to true** at that point during code execution.

```
1  #include <assert.h>
2  void main() {
3      int x_axis = 5;
4      int y_axis = 1;
5      // assertion succeeds
6      assert(x_axis > 0 && y_axis > 0);
7      plot(x_axis, y_axis);
8      int y_axis = x - 5;
9      // assertion fails and program crashes
10     assert(x_axis > 0 && y_axis > 0);
11 }
```

Assertions as Specifications in Software Verification (Example)

- Assertions are program statements used to **test assumptions** made by software designers/programmers/testers.
- An assertion is a expression/predicate that **always should evaluate to true** at that point during code execution.

```c
#include <assert.h>
void main() {
    int x_axis = 5;
    int y_axis = 1;
    // assertion succeeds
    assert(x_axis > 0 && y_axis > 0);
    plot(x_axis, y_axis);
    int y_axis = x - 5;
    // assertion fails and program crashes
    assert(x_axis > 0 && y_axis > 0);
}
```

```c
void display_number(int* myInt) {
  assert(myInt != nullptr);
  printf("%d", *myInt);
}
int main () {
  int myptr = 5;
  int* first_ptr = &myptr;
  int* second_ptr = nullptr;
  // assertion succeeds
  display_number(first_ptr);
  // assertion fails and program crashes
  display_number(second_ptr);
}
```

Assertion-Based Software Verification

Assertions can be seen as partial software specifications that

- help a programmer **read the code**
- help the program **detect its own defects**
- help catch errors earlier and **pinpoint sources of errors**

Assertion-Based Software Verification

Assertions can be seen as partial software specifications that

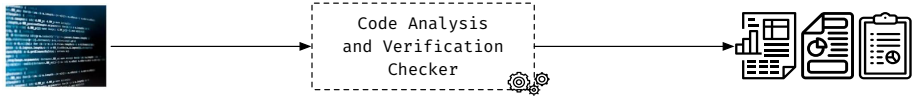
- help a programmer **read the code**
- help the program **detect its own defects**
- help catch errors earlier and **pinpoint sources of errors**

Assertions are typically used in the following scenarios.

- Software **developers** can add assertions during their programming or implementation to verify their expected results.
- Software **testers** can add assertions as a part of the unit testing process.
- Project **managers** or third parties can add assertions in the middle or end of a program execution to verify and understand code bases.

The Project of This Subject

Goal of this subject: develop your own software verification tool in 12 weeks.
More concretely: develop a static symbolic execution engine in C++ to verify C programs with assertions at compile time.



The Project of This Subject

Source Code

Assignment 1: Information Flow Tracking

Assignment 2: Symbolic Execution

Assignment 3: Abstract Interpretation

Static Analysis Engine

1. Tainted Information Flow

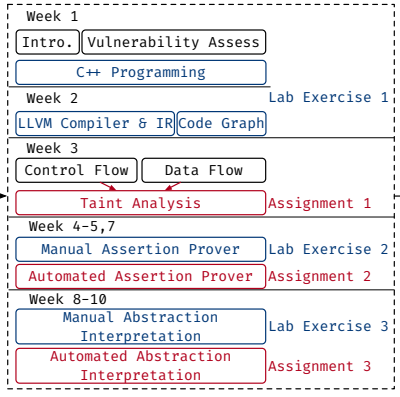
2. Assertion Checking

3. Buffer Overflow

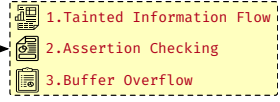
Analysis Report

The Project of This Subject

Source Code



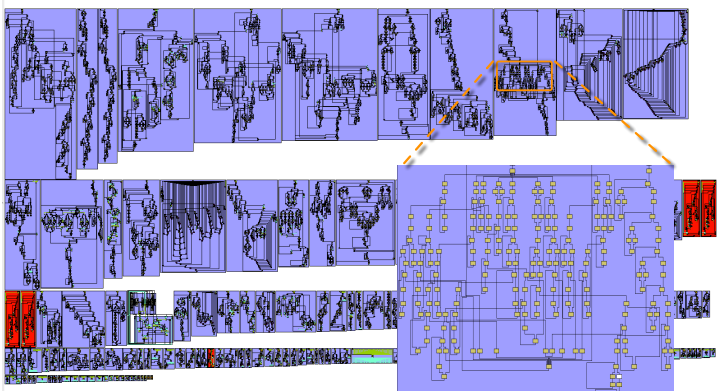
Static Analysis Engine



Analysis Report

How about real-world large programs?

Whole-Program CFG of 300.twolf (20.5K lines of code)

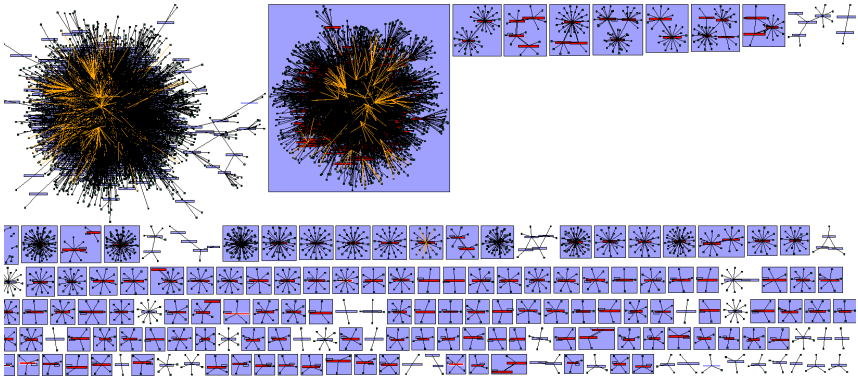


#functions: 194

#pointers: 20773 #loads/stores: 8657 Costly to reason about flow of values on CFGs!

How about real-world large programs?

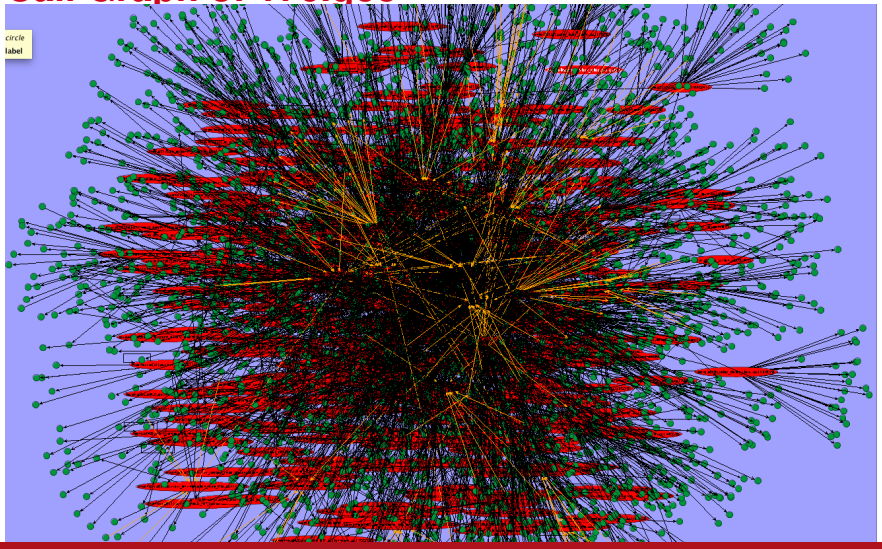
Call Graph of 176.gcc (230.5K lines of code)



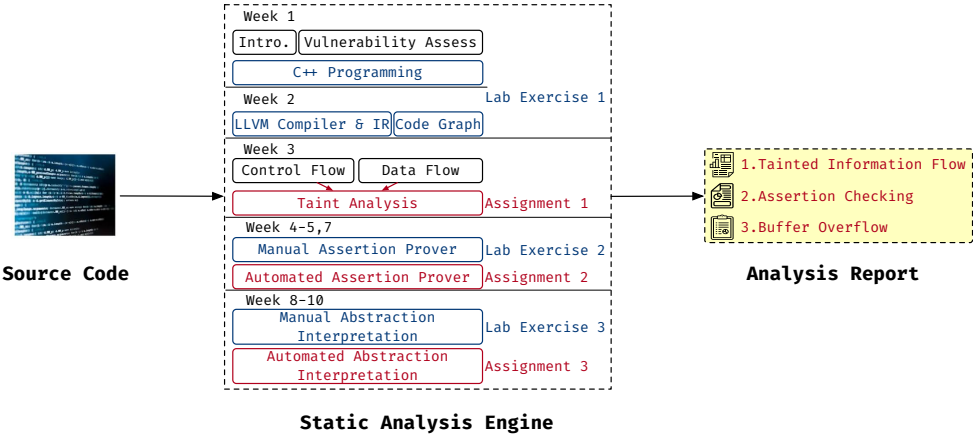
#functions: 2256 #pointers: 134380 #loads/stores: 51543

Costly to reason about flow of values on CFGs!

Call Graph of 176.gcc

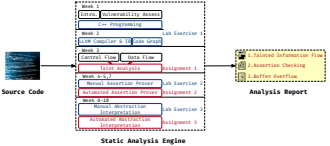


The Project of This Subject

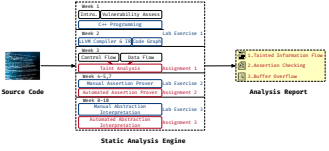


The Project of This Subject

The project sounds complicated?



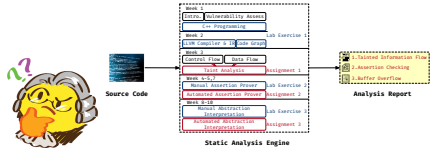
The Project of This Subject



The project sounds complicated?

- Do I need to implement it from scratch?

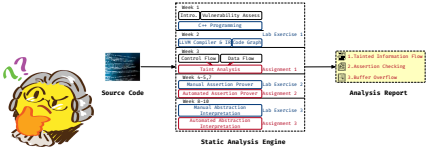
The Project of This Subject



The project sounds complicated?

- Do I need to implement it from scratch?
  - **No**, you will implement a lightweight tool based on the open-source framework SVF (`https://github.com/SVF-tools/SVF`)
- How many lines of code do I need to write?

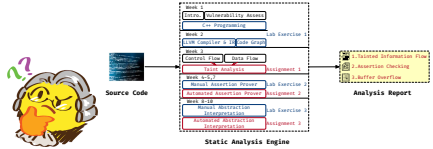
The Project of This Subject



The project sounds complicated?

- Do I need to implement it from scratch?
  - **No**, you will implement a lightweight tool based on the open-source framework SVF (`https://github.com/SVF-tools/SVF`)
- How many lines of code do I need to write?
  - **1,000 lines** of core code **in total** for all the three assignments
- Really? What are the challenges then?

The Project of This Subject



The project sounds complicated?

- Do I need to implement it from scratch?
  - **No**, you will implement a lightweight tool based on the open-source framework SVF (`https://github.com/SVF-tools/SVF`)
- How many lines of code do I need to write?
  - **1,000 lines** of core code **in total** for all the three assignments
- Really? What are the challenges then?
  - Good programming and debugging skills.
  - Understanding of basic compiler principles and code graphs.
  - Knowledge of taint analysis, symbolic execution, and abstract interpretation.
  - **Please do attend each class** to make sure you can keep up!

Vulnerability Assessment and Secure Coding

(Week 1)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Software Vulnerability

Wiki's definition: A vulnerability is a weakness which can be exploited by a threat actor, such as an attacker, to cross privilege boundaries (i.e. perform unauthorized actions) within a computer system

Software Vulnerability

Wiki's definition: A vulnerability is a weakness which can be exploited by a threat actor, such as an attacker, to cross privilege boundaries (i.e. perform unauthorized actions) within a computer system

Causes of Software Vulnerabilities:

- High-level design flaws
- Insecure coding (**This subject**)

Common Types of Software Vulnerabilities

- Memory safety errors
  - Memory Leaks
  - Null pointer dereferences
  - Dangling pointers and use-after-frees
  - Buffer overflows
- Arithmetic errors
  - Integer overflows
  - Division by zero
- Tainted inputs
  - Tainted information flow
  - Code injection
  - Format string
  - SQL injection
- Side-channel attacks
  - Timing attacks

Let us take a look at examples of the above vulnerabilities, how to fix them and implement more secure programming practices (e.g., using assertions)

Memory Leaks (A Vulnerable Example)

A memory leak occurs when dynamically allocated memory is never freed along a program execution path.

Memory Leaks (A Vulnerable Example)

A memory leak occurs when dynamically allocated memory is never freed along a program execution path.

```
1  typedef struct Node {
2      int data;
3      struct Node *next;
4  } Node;
5  typedef struct List {
6      struct Node *head;
7  } List;
8
9  List* create_list(int num){
10     List* list = new List();
11     Node* current = list->head;
12     for(i = 0; i < num; i++){
13         current = new Node();
14         current = current->next;
15     }
16     return list;
17 }
```

```
1  void free_list(List *l) {
2      Node* current = l->head;
3      while (current != NULL) {
4          node *next = current->next;
5          delete current;
6          current = next;
7      }
8  }
9
10 int main(){
11   List* l = create_list(10); // create a list of 10 nodes
12   // ... do something with the list
13   free_list(l); // deallocate the memory for the list
14 }
```

Memory Leaks (A Vulnerable Example)

A memory leak occurs when dynamically allocated memory is never freed along a program execution path.

```
1  typedef struct Node {
2      int data;
3      struct Node *next;
4  } Node;
5  typedef struct List {
6      struct Node *head;
7  } List;
8
9  List* create_list(int num){
10     List* list = new List();
11     Node* current = list->head;
12     for(i = 0; i < num; i++){
13         current = new Node();
14         current = current->next;
15     }
16     return list;
17 }
```

```
1  void free_list(List *l) {
2      Node* current = l->head;
3      while (current != NULL) {
4          node *next = current->next;
5          delete current;
6          current = next;
7      }
8  }
9
10 int main(){
11   List* l = create_list(10); // create a list of 10 nodes
12   // ... do something with the list
13   free_list(l); // deallocate the memory for the list
14 }
```

Do we have a bug in the above code?
How do we fix the bug?

Memory Leaks (A Secure Example)

```
1  typedef struct Node {
2      int data;
3      struct Node *next;
4  } Node;
5  typedef struct List {
6      struct Node *head;
7  } List;
8
9  List* create_list(int num){
10     List* list = new List();
11     Node* current = list->head;
12     for(i = 0; i < num; i++){
13         current = new Node();
14         current = current->next;
15     }
16     return list;
17 }
```

```
1  void free_list(List *l) {
2      Node* current = l->head;
3      while (current != NULL) {
4          node *next = current->next;
5          delete current;
6          current = next;
7      }
8      delete l;
9  }
10
11 int main(){
12   List* l = create_list(10); // create a list of 10 nodes
13   // ... do something with the list
14   free_list(l); // deallocate the memory for the list
15 }
```

Memory Leaks (A Secure Example)

```
1  typedef struct Node {
2      int data;
3      struct Node *next;
4  } Node;
5  typedef struct List {
6      struct Node *head;
7  } List;
8
9  List* create_list(int num){
10     List* list = new List();
11     Node* current = list->head;
12     for(i = 0; i < num; i++){
13         current = new Node();
14         current = current->next;
15     }
16     return list;
17 }
```

```
1  void free_list(List *l) {
2      Node* current = l->head;
3      while (current != NULL) {
4          node *next = current->next;
5          delete current;
6          current = next;
7      }
8      delete l;
9  }
10
11 int main(){
12   List* l = create_list(10); // create a list of 10 nodes
13   // ... do something with the list
14   free_list(l); // deallocate the memory for the list
15   l = nullptr;  // avoid misuse later
16 }
```

Null Pointer Dereferences (A Vulnerable Example)

A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.

```
1  struct Student{
2  int id;
3  char* name;
4  } Student;
5  Student students[10]= {...};
6
7  Student* findStuRecord(int sID){
8      for(int i = 0; i < 10; i++){
9          if(students[i].id == sID)
10             return &students[i];
11     }
12     return nullptr;
13 }
```

```
1 int main(int argc, char **argv){
2     int stuID;
3     scanf( "%d%c", &stuID);
4     Student* student = findStuRecord(stuID);
5     printf("%s\n", student->name);
6 }
```

Null Pointer Dereferences (A Vulnerable Example)

A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.

```
1  struct Student{
2  int id;
3  char* name;
4  } Student;
5  Student students[10]= {...};
6
7  Student* findStuRecord(int sID){
8      for(int i = 0; i < 10; i++){
9          if(students[i].id == sID)
10             return &students[i];
11     }
12     return nullptr;
13 }
```

```
1 int main(int argc, char **argv){
2     int stuID;
3     scanf( "%d%c", &stuID);
4     Student* student = findStuRecord(stuID);
5     printf("%s\n", student->name);
6 }
```

Do we have a bug in the above code? If so, where should we put the assertion to stop the bug?

Null Pointer Dereferences (A Secure Example)

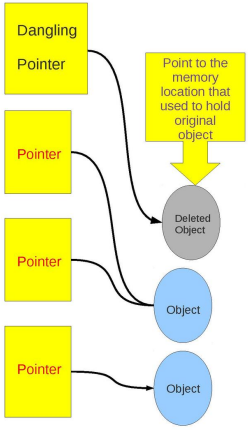
A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.

```
1  struct Student{
2  int id;
3  char* name;
4  } Student;
5  students[10]= {...};
6
7  Student* findStuRecord(int sID){
8      for(int i = 0; i < 10; i++){
9          if(students[i].id == sID)
10             return students[i];
11     }
12     return NULL;
13 }
```

```
1 int main(int argc, char **argv){
2     int stuID;
3     scanf( "%d%c", &stuID);
4     Student* student = findStuRecord(stuID);
5     assert(student!=nullptr);
6     printf("%s\n", student->name);
7 }
```

Dangling references and use-after-frees

Dangling references are references that do not resolve to a valid memory object (e.g., caused by a use-after-free).



Dangling references and use-after-frees (A Vulnerable Example)

Dangling references are references that do not resolve to a valid memory object (e.g., caused by a use-after-free).

```
1  char* ptr = (char*)malloc(SIZE);
2  ...
3  if (err) {
4      abrt = 1;
5      free(ptr);
6  }
7  ...
8  if (abrt) {
9      logError("operation aborted before commit", ptr);
10 }
```

Do we have a bug in the above code? If so, how should we fix the bug?

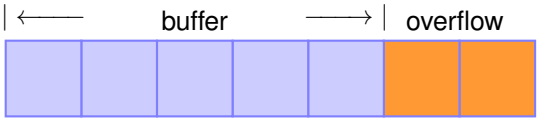
Dangling references and use-after-frees (A Vulnerable Example)

Dangling references and wild references are references that do not resolve to a valid destination

```
1  char* ptr = (char*)malloc(SIZE);
2  ...
3  if (err) {
4      abrt = 1;
5      free(ptr);
6      ptr = nullptr;
7  }
8  ...
9  if (abrt) {
10     assert(ptr != nullptr);
11     logError("operation aborted before commit", ptr);
12 }
```

Buffer Overflows

A buffer overflow occurs when the volume of data exceeds the storage capacity of the memory buffer, so that the program attempting to write the data to the buffer overwrites adjacent memory locations.



Buffer Overflows (A Vulnerable Example)

A buffer overflow occurs when the volume of data exceeds the storage capacity of the memory buffer, so that the program attempting to write the data to the buffer overwrites adjacent memory locations.

```
1  void bufferRead() {
2      int n = 0;
3      int ret = scanf("%d", &n);
4      if (ret != 1 || n > 100) {
5          return;
6      }
7      char *p = (char *)malloc(n);
8      int y = n;
9      if (p == NULL) return;
10     p[y] = 'a';
11     free(p);
12 }
```

Do we have a bug in the above code? If so, where is the bug?

Buffer Overflows (A Secure Example)

A buffer overflow occurs when the volume of data exceeds the storage capacity of the memory buffer, so that the program attempting to write the data to the buffer overwrites adjacent memory locations.

```
1  void bufferRead() {
2      int n = 0;
3      int ret = scanf("%d", &n);
4      if (ret != 1 || n > 100) {
5          return;
6      }
7      char *p = (char *)malloc(n);
8      int y = n;
9      if (p == NULL) return;
10     assert(y < n);
11     p[y] = 'a';
12     free(p);
13 }
```

Integer Overflows

An integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of digits – either higher than the maximum or lower than the minimum representable value.

Integer Overflows

An integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of digits – either higher than the maximum or lower than the minimum representable value.

Typical binary register widths for unsigned integers include

- 4 bits: maximum representable value 2^4 - 1 = 15
- 8 bits: maximum representable value 2^8 - 1 = 255
- 16 bits: maximum representable value 2^{16} - 1 = 65,535
- 32 bits: maximum representable value 2^{32} - 1 = 4,294,967,295 (the most common width for personal computers as of 2005),
- 64 bits: maximum representable value 2^{64} - 1 = 18,446,744,073,709,551,615 (the most common width for personal computer CPUs, as of 2017),
- 128 bits: maximum representable value 2^{128} - 1 = 340,282,366,920,938,463,463,374,607,431,768,211,455

Integer Overflows

In C/C++, overflows of **signed integer** will cause **undefined behavior** but overflows of unsigned integer will not. For unsigned integers, when the value exceeds the maximum value (2^n for some n), the result is reduced to that value modulo 2^n.

Here are some of C's various integer types and the values (in standard library):

- INT_MIN (the minimum value for an integer): -2,147,483,648 bits
- INT_MAX (the maximum value for an integer): +2,147,483,647 bits
- UINT_MAX (the maximum value for an unsigned integer): 4,294,967,295 bits

Integer Overflows

In C/C++, overflows of **signed integer** will cause **undefined behavior** but overflows of unsigned integer will not. For unsigned integers, when the value exceeds the maximum value (2^n for some n), the result is reduced to that value modulo 2^n.

Here are some of C's various integer types and the values (in standard library):

- INT_MIN (the minimum value for an integer): -2,147,483,648 bits
- INT_MAX (the maximum value for an integer): +2,147,483,647 bits
- UINT_MAX (the maximum value for an unsigned integer): 4,294,967,295 bits
- UINT_MAX + 1 = ?
- UINT_MAX + 2 = ?
- UINT_MAX + 3 = ?

Integer Overflows

In C/C++, overflows of **signed integer** will cause **undefined behavior** but overflows of unsigned integer will not. The resulting unsigned integer type is reduced modulo to the number that is one greater than the largest value that can be represented by the resulting type (modulo power of two, i.e., 2^n where n is No. of bits).

Here are some of C's various integer types and the values (in standard library):

- INT_MIN (the minimum value for an integer): -2,147,483,648 bits
- INT_MAX (the maximum value for an integer): +2,147,483,647 bits
- UINT_MAX (the maximum value for an unsigned integer): 4,294,967,295 bits
- UINT_MAX + 1 == 0
- UINT_MAX + 2 == 1
- UINT_MAX + 3 == 2

Integer Overflows[2]

Compilers may exploit undefined behavior and optimize when there is an overflow of a signed integer.

Integer Overflows[2]

Compilers may exploit undefined behavior and optimize when there is an overflow of a signed integer.

```
1 signed int x ;
2 if(x > x + 1)
3 {
4     // Handle potential overflow
5 }
```

```
1 if (INT_MAX + 1 < 0){
2     // Overlooked
3 }
```

Here since a signed integer overflow is not defined, compiler is free to assume that it may never happen and hence it can optimize away the ”if” blocks.

```
1 signed i = 1;
2 while (i > 0){
3     i *= 2;
4 }
```

The above behavior is undefined and when compiled under ‘-O3‘ using GCC (version 4.7.2 on Debian 4.7.2-5). It performs branch simplifications, producing an infinite loop.

Integer Overflows (A Vulnerable Example)

```
1 int nresp = packet_get_int();
2
3 if (nresp > 0) {
4  response = xmalloc(nresp*sizeof(char*));
5
6  for (i = 0; i < nresp; i++)
7     response[i] = packet_get_string(NULL);
8 }
```

Do we have a bug in the above code? If so, where is the bug?

Integer Overflows (A Vulnerable Example)

```
1 int nresp = packet_get_int();
2
3 if (nresp > 0) {
4  response = xmalloc(nresp*sizeof(char*));
5
6  for (i = 0; i < nresp; i++)
7     response[i] = packet_get_string(NULL);
8 }
```

Do we have a bug in the above code? If so, where is the bug?

- An integer overflow that leads to a buffer overflow found in an older version of OpenSSH (3.3):
- If nresp is greater than or equal to 1073741824 and sizeof(char*) is 4 (which is typical of 32-bit systems), then nresp*sizeof(char*) results in an overflow. Therefore, xmalloc() receives and allocates a small buffer. The subsequent loop causes a heap buffer overflow, which may, in turn, be used by an attacker to execute arbitrary code.

Integer Overflows (A Secure Example)

```
1 int nresp = packet_get_int();
2  assert(nresp < userDefinedSize);
3 if (nresp > 0) {
4  response = xmalloc(nresp*sizeof(char*));
5
6  for (i = 0; i < nresp; i++)
7   response[i] = packet_get_string(NULL);
8 }
```

Make sure the size received is under a user budget.

Integer Overflows (A Secure Example)

```
1 int nresp = packet_get_int();
2  assert(nresp < userDefinedSize);
3 if (nresp > 0) {
4  response = xmalloc(nresp*sizeof(char*));
5
6  for (i = 0; i < nresp; i++)
7   response[i] = packet_get_string(NULL);
8 }
```

Make sure the size received is under a user budget.

```
1 int nresp = packet_get_int();
2
3 if (nresp > 0) {
4  assert(nresp <= userDefinedSize/sizeof(char
  *));
5  response = xmalloc(nresp*sizeof(char*));
6
7  for (i = 0; i < nresp; i++)
8   response[i] = packet_get_string(NULL);
9 }
```

Make sure the malloc can safely allocate memory under a user budget.

Division by Zero (A Vulnerable Example)

The product divides a value by zero. The C standard (C11 6.5.5) states that dividing by zero has undefined behavior for either integer or floating-point operands. It can cause program crash or miscompilation by compilers.

```
1 unsigned computeAverageResponseTime (unsigned totalTime, unsigned numRequests)
2 {
3     return totalTime / numRequests;
4 }
```

Is the above code secure?

Division by Zero (A Secure Example)

The product divides a value by zero. The C standard (C11 6.5.5) states that dividing by zero has undefined behavior for either integer or floating-point operands. It can cause program crash or miscompilation by compilers.

```
1 unsigned computeAverageResponseTime (unsigned totalTime, unsigned numRequests)
2 {
3     assert(numRequests > 0);
4     return totalTime / numRequests;
5 }
```

Tainted Information Flow (A Vulnerable Example)

Malicious user inputs may cause unexpected program behaviors, information leakage or attacks when executing a target program.

```
1  void main(int argc, char **argv)
2  {
3      char *pMsg = packet_get_string();
4      ParseMsg((LOGIN_MSG_BODY *)pMsg);
5  }
6
7  void ParseMsg(LOGIN_MSG_BODY *stLoginMsgBody)
8  {
9      for (int ulIndex = 0; ulIndex < stLoginMsgBody->usLoginPwdLen; ulIndex++) {
10         // do something
11     }
12 }
```

Tainted Information Flow (A Vulnerable Example)

Malicious user inputs may cause unexpected program behaviors, information leakage or attacks when executing a target program.

```
1  void main(int argc, char **argv)
2  {
3      char *pMsg = packet_get_string();
4      ParseMsg((LOGIN_MSG_BODY *)pMsg);
5  }
6
7  void ParseMsg(LOGIN_MSG_BODY *stLoginMsgBody)
8  {
9      for (int ulIndex = 0; ulIndex < stLoginMsgBody->usLoginPwdLen; ulIndex++) {
10         // do something
11     }
12 }
```

Do we have a bug in the above code? If so, where is the bug?

Tainted Information Flow (A Secure Example)

```
1  void main(int argc, char **argv)
2  {
3      char *pMsg = packet_get_string();
4      ParseMsg((LOGIN_MSG_BODY *)pMsg);
5  }
6
7  void ParseMsg(LOGIN_MSG_BODY *stLoginMsgBody)
8  {
9      assert(stLoginMsgBody->usLoginPwdLen < userDefinedSize);
10     for (int ulIndex = 0; ulIndex < stLoginMsgBody->usLoginPwdLen; ulIndex++) {
11         // do something
12     }
13 }
```

User input can cause long loops to hang the target program.

Code Injection (A Vulnerable Example)

Code injection is the exploitation of a computer bug that is caused by processing invalid data. The result of successful code injection can be disastrous, for example, by disclosing confidential information.

```
1  string user_id, command;
2  command = "cat user_info/";
3  cin >> user_id;  // user_id is an integer
4  system(command + user_id);
```

Is the above code secure?

Code Injection (A Vulnerable Example)

Code injection is the exploitation of a computer bug that is caused by processing invalid data. The result of successful code injection can be disastrous, for example, by disclosing confidential information.

```
1  string user_id, command;
2  command = "cat user_info/";
3  cin >> user_id;  // user_id is an integer
4  system(command + user_id);
```

Is the above code secure?

- Before system(), we did not check the legality of user_id. first, if x is a piece of dangerous code, it will be executed in system() and return harmful results.
- For example, if user types ”05 && ipconfig”, it will disclose ip address.

Code Injection (A Secure Example)

To detect the bug, we need to make sure after appending $user_id will not generate another command.

```
1  string user_id, command;
2  command = "cat user_info/";
3  cin>>user_id; // what if user input "05 && ipconfig"
4  assert(isdigit(user_id));
5  system(command + user_id);
```

In the assert call, we should check whether user_id consists entirely of numeric characters, so that another malicious command would not be injected.

Format String

The Format String exploit occurs when the submitted data of an input string is evaluated as a command by the application.

```
1 #include  <stdio.h>
2 void main(int argc, char **argv){
3     printf("%s\n", argv[1]);
4     printf(argv[1]);
5 }
```

Which `printf` is safe and which `printf` is vulnerable? why?

Format String

The Format String exploit occurs when the submitted data of an input string is evaluated as a command by the application.

```
1 #include  <stdio.h>
2 void main(int argc, char **argv){
3     printf("%s\n", argv[1]);
4     printf(argv[1]);
5 }
```

Which `printf` is safe and which `printf` is vulnerable? why?

- The line `printf("%s", argv[1]);` is safe, if you compile the program and run it:
  - `./example "Hello World %s%s%s%s%s%s"`

Format String

The Format String exploit occurs when the submitted data of an input string is evaluated as a command by the application.

```
1 #include  <stdio.h>
2 void main(int argc, char **argv){
3     printf("%s\n", argv[1]);
4     printf(argv[1]);
5 }
```

Which `printf` is safe and which `printf` is vulnerable? why?

- The line `printf("%s", argv[1]);` is safe, if you compile the program and run it:
  - `./example "Hello World %s%s%s%s%s%s%s"`
  - Output: `"Hello World %s%s%s%s%s%s%s"`

Format String

The Format String exploit occurs when the submitted data of an input string is evaluated as a command by the application.

```
1 #include  <stdio.h>
2 void main(int argc, char **argv){
3     printf("%s\n", argv[1]);
4     printf(argv[1]);
5 }
```

Which `printf` is safe and which `printf` is vulnerable? why?

- The line `printf(argv[1]);` in the example is vulnerable
- `./example "Hello World %s%s%s%s%s%s"`
- Output: program crash.
- The `printf` in the second line will interpret the `%s%s%s%s%s%s` in the input string as a reference to string pointers. It will try to interpret every `%s` as a pointer to a string, which will get to an invalid address, and attempting to access it will likely cause the program to crash

Format String

The Format String exploit occurs when the submitted data of an input string is evaluated as a command by the application.

```
1 #include  <stdio.h>
2 void main(int argc, char **argv){
3     printf("%s\n", argv[1]);
4     printf(argv[1]);
5 }
```

- It can be more than a crash! An attacker can also use this to get information. For example, running: `./example "Hello World %p %p %p %p %p %p"`
- First `printf` output: `Hello World %p %p %p %p %p %p`
- Second `printf` output: `Hello World 000E133E 000E133E 0057F000 CCCCCCCC CCCCCCCC CCCCCCCC`
- Causing possible memory address leakage and exploit[3]

SQL Injection (A Vulnerable Example)

The attacker can add additional SQL statements at the end of the query statements defined in advance (e.g., in the web application).

```
1 txtUserId = getRequestString("UserId");
2 txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

Is the above code vulnerable? Why?

SQL Injection (A Vulnerable Example)

The attacker can add additional SQL statements at the end of the query statements defined in advance (e.g., in the web application).

```
1 txtUserId = getRequestString("UserId");
2 txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

Is the above code vulnerable? Why?

- The original purpose of the code was to create an SQL statement to select a user, with a given user id.
- If there is nothing to prevent a user from entering ”wrong” input, the user can enter some ”smart” input like this: `"105 OR 1=1"`.
- ”OR 1==1” will make logical expression after WHERE always true. Then the sql database will return all the items from the Users table.

SQL Injection(A Secure Example)

To detect the bug, we need to verify the user's input.

```
1 txtUserId = "105 OR 1=1";
2 assert(isdigit(txtUserId));
3 txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

Assuming that user id is an integer, we should assert the user's input is exactly an integer.

Side-Channel Attacks (A Vulnerable Example)

```
1 bool insecureCmp(char *ca, char *cb, int length)
2 {
3   for (int i = 0; i < length; i++)
4     if (ca[i] != cb[i])
5       return false;
6   return true;
7 }
```

- The above C code demonstrates a typical insecure string comparison which stops testing as soon as a character doesn't match.

Side-Channel Attacks (A Vulnerable Example)

```
1 bool insecureCmp(char *ca, char *cb, int length)
2 {
3   for (int i = 0; i < length; i++)
4     if (ca[i] != cb[i])
5       return false;
6   return true;
7 }
```

- The above C code demonstrates a typical insecure string comparison which stops testing as soon as a character doesn't match.
- Assume a computer that takes 1 ms to check each character of a password using the above function to validate passwords.
- If the attacker guesses a completely wrong password, it will take 1 ms. Once they get one character right, it takes 2 ms. Two correct characters take 3 ms, and so on.

Side-Channel Attacks (A Secure Example)

```
1 bool insecureCmp(char *ca, char *cb, int length)
2 {
3   for (int i = 0; i < length; i++)
4     if (ca[i] != cb[i])
5       return false;
6   return true;
7 }
```

```
1 # String comparison time: 1ms
2 'aaaaaaaaaaaaaaaa' == 'V1cHt2S67DADJIm9s'
3
4 # String comparison time: 1ms
5 'baaaaaaaaaaaaaaa' == 'V1cHt2S67DADJIm9s'
6
7 # String comparison time: 2ms
8 'Vaaaaaaaaaaaaaaa' == 'V1cHt2S67DADJIm9s'
9
10 # String comparison time: 3ms
11 'V1aaaaaaaaaaaaaaa' == 'V1cHt2S67DADJIm9s'
12 # ...
```

Side-Channel Attacks (A Secure Example)

```
1 bool constTimeCmp(char *ca, char *cb, int length)
2 {
3   bool result = true;
4   for (int i = 0; i < length; i++)
5     result &= (ca[i] == cb[i]);
6   return result;
7 }
```

- The above secure version runs in constant-time by testing all characters and using a bitwise operation to accumulate the result.

Side-Channel Attacks (A Secure Example)

```
1 bool constTimeCmp(char *ca, char *cb, int length)
2 {
3   bool result = true;
4   int i;
5   for (i = 0; i < length; i++)
6     result &= (ca[i] == cb[i]);
7   assert(length == i);  //  make sure the comparison times is proportional to the length.
8   return result;
9 }
```

- Make sure the each time **calling the comparison function always costs the same time (constant-time comparison)**
- The above secure version runs in constant-time by testing all characters and using a bitwise operation to accumulate the result.

What's Next?

- Configure your programming environment
  `https://github.com/SVF-tools/Software-Security-Analysis/wiki/Installation-of-Docker,-VSCode-and-its-extensions`
  - Write and run your program in a docker container (virtual machine) in VSCode so you can debug and run your programs on top of any operating system.
- Write a hello world C++ program within the Docker container via VSCode.
- Revisit and practice C++ programming (more about C++ programming will be coming in our lab exercise)
- Start working on the first set of quizzes.