Introduction to Software Security Analysis

(Week 1)

Yulei Sui

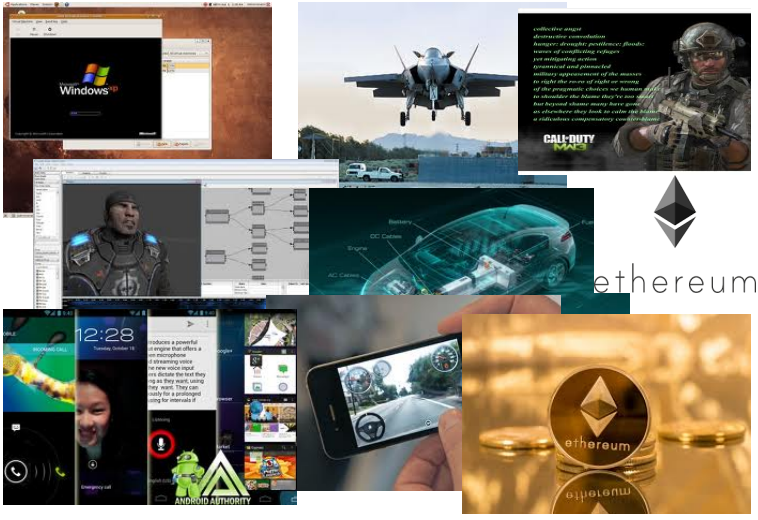
School of Computer Science and Engineering

University of New South Wales, Australia

Outline

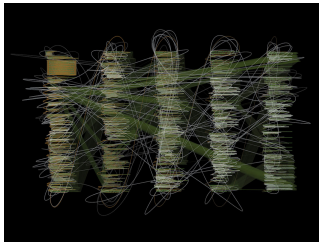
- Background and Introduction to Software Analysis and Verification
- Course Project Structure (Labs and Assignments)
- Vulnerability Assessment and Secure Coding.

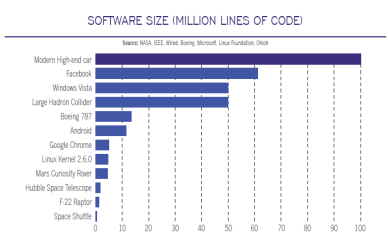
Software Is Everywhere

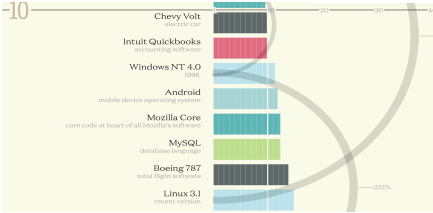


Modern System Software

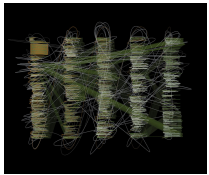
– Extremely Large and Complex

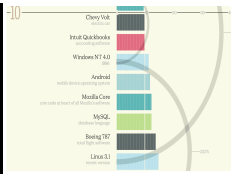






Software Becomes More Buggy





More Complex!

Memory Leaks

Buffer Overflows

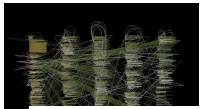
Null Pointers

Use-After-Frees

Data-races

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!

Data-races

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

Code Review by Developers

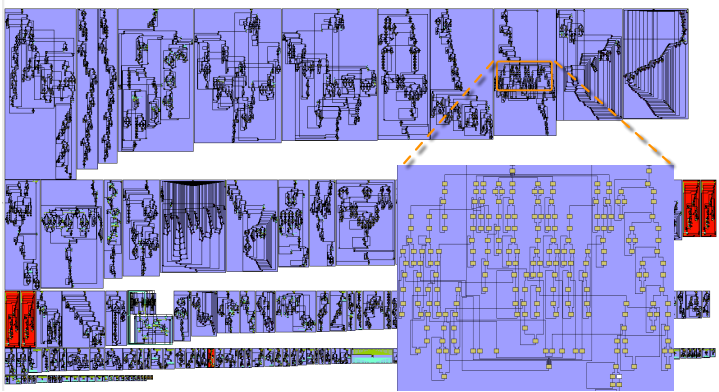
However ...

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 `twolf` (20.5K lines of code)

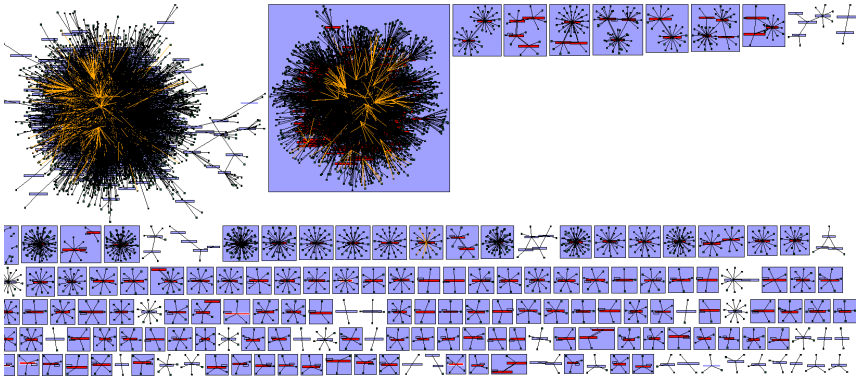


#functions: 194

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

How about real-world large programs?

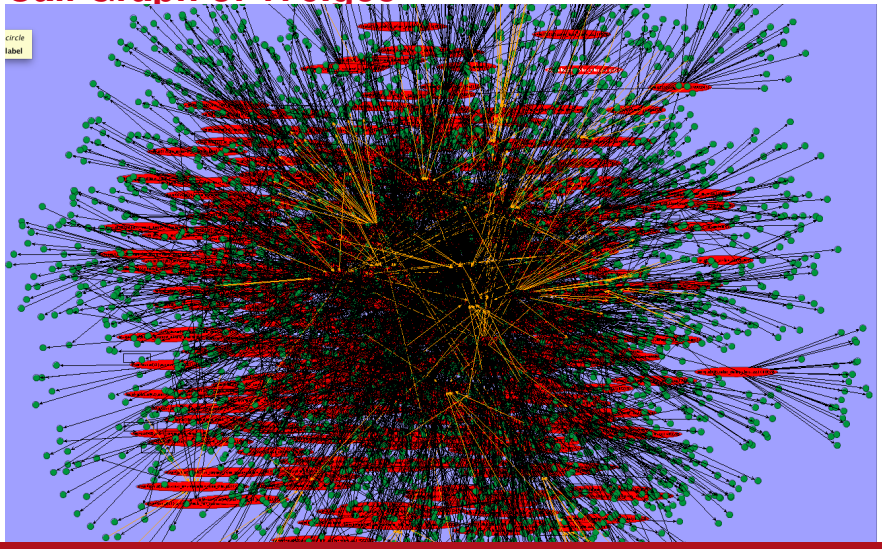
Call Graph of `gcc` (230.5K lines of code)



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

Costly to reason about flow of values on CFGs!

Call Graph of 176.gcc



Automated Code Analysis and Verification

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

- **Software Analysis** (Week 1-3, Assignment 1)
  - Aim to ***find the existence of bugs*** (If a path exists where a bug may be triggered, report that bug)
- **Software Verification** (Week 4-5,7-10, Assignments 2 and 3)
  - Aim to ***prove the absence of bugs*** (For all paths, user specification should be satisfied and no bug should be triggered)

Automated Code Analysis and Verification

- Software analysis and verification are 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 course**)
  - Code specs (e.g., return a sorted list and free of memory errors):
    - Assertions and pre/postconditions in Hoare logic (design by contract)
    - Passing user-provided test cases
    - No crashes and free of memory errors
    - ...

How to Perform Code Analysis and Verification?

How to Perform Code 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 Code 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 course**)
  - **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 Code 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 course**)
  - **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**

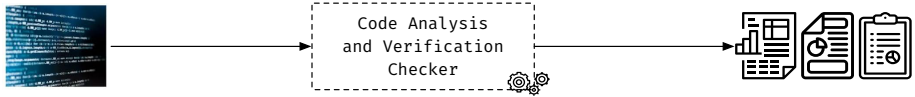
How to Perform Code Analysis and Verification?

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 course**)
  - **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**

The Project of This Course

Goal of this course: develop your own software verification tool in 10-week time.
More concretely: develop a static analysis engine in C++ to analyze and verify C programs for bug detection at compile time.



The Project of This Course

Assignment 1: Information Flow Tracking

Assignment 2: Symbolic Execution

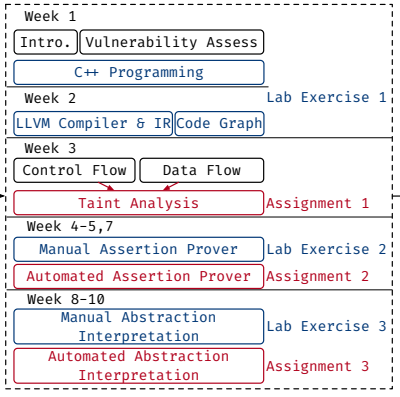
Assignment 3: Abstract Interpretation

1. Tainted Information Flow
2. Assertion Checking
3. Buffer Overflow

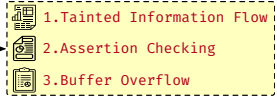
Source Code **Static Analysis Engine** **Analysis Report**

The Project of This Course

Source Code



Static Analysis Engine



Analysis Report

The Project of This Course

Source Code

Week 1
Intro.
Vulnerability Assess
C++ Programming

Week 2
LLVM Compiler & IR
Code Graph
Lab Exercise 1

Week 3
Control Flow
Data Flow
Taint Analysis
Assignment 1

Week 4-5,7
Manual Assertion Prover
Lab Exercise 2
Automated Assertion Prover
Assignment 2

Week 8-10
Manual Abstraction Interpretation
Lab Exercise 3
Automated Abstraction Interpretation
Assignment 3

Static Analysis Engine

1.Tainted Information Flow
2.Assertion Checking
3.Buffer Overflow

Analysis Report

The Project of This Course

The project sounds complicated?

The Project of This Course

The project sounds complicated?

- Do I need to implement it from scratch?

The Project of This Course

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`)
  - **SVF**, an impactful code analysis framework developed and maintained by UNSW for 10+ years (ICSE, OOPSLA and SAS Distinguished Paper awards).
- How many lines of code do I need to write?

The Project of This Course

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`)
  - **SVF**, an impactful code analysis framework developed and maintained by UNSW for 10+ years (ICSE, OOPSLA and SAS Distinguished Paper awards).
- How many lines of code do I need to write?
  - **2,000 lines** of core code **in total** for all the assessments
- Really? What are the challenges then?

The Project of This Course

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`)
  - **SVF**, an impactful code analysis framework developed and maintained by UNSW for 10+ years (ICSE, OOPSLA and SAS Distinguished Paper awards).
- How many lines of code do I need to write?
  - **2,000 lines** of core code **in total** for all the assessments
- Really? What are the challenges then?
  - Good programming and debugging skills.
  - Understanding of basic compiler principles,
  - Knowledge of taint analysis, symbolic execution, and abstract interpretation.
  - **Please do attend each lecture and lab** 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

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     list->head = new Node();
12     Node* current = list->head;
13     for(i = 0; i < num; i++){
14         current = new Node();
15         current = current->next;
16     }
17     return list;
18 }
```

```
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     list->head = new Node();
12     Node* current = list->head;
13     for(i = 0; i < num; i++){
14         current = new Node();
15         current = current->next;
16     }
17     return list;
18 }
```

```
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     list->head = new Node();
12     Node* current = list->head;
13     for(i = 0; i < num; i++){
14         current->next = new Node();
15         current = newNode;
16     }
17     return list;
18 }
```

```
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
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     list->head = new Node();
12     Node* current = list->head;
13     for(i = 0; i < num; i++){
14         current->next = new Node();
15         current = newNode;
16     }
17     return list;
18 }
```

```
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)

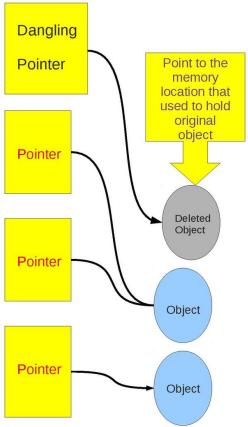
A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.

```
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 / 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?

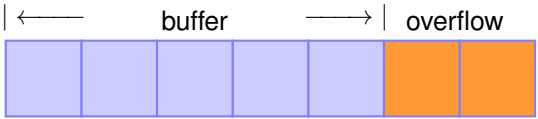
Dangling References / Use-After-Frees (A Vulnerable Example)

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**. Overflows of **unsigned integer** will cause a **wraparound**. 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**. Overflows of **unsigned integer** will cause a **wraparound**. 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

Compilers may exploit undefined behavior and optimize when there is an overflow of a signed integer. Hence, the compiler may make an incorrect or undesired optimization, producing code that you may not want!

Integer Overflows

Compilers may exploit undefined behavior and optimize when there is an overflow of a signed integer. Hence, the compiler may make an incorrect or undesired optimization, producing code that you may not want!

```
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.

Integer Overflows

Compilers may exploit undefined behavior and optimize when there is an overflow of a signed integer. Hence, the compiler may make an incorrect or undesired optimization, producing code that you may not want!

```
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){          //Be careful using signed integers as bounds in loops and branches!
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.

https://stackoverflow.com/questions/23889022/gcc-optimizations-based-on-integer-overflow

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

https://cs155.stanford.edu/papers/formatstring-1.2.pdf

https://robin-sandhu.medium.com/understanding-the-format-string-vulnerability-b2957630d886

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 }
```

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 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 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?

- (1) Configure your programming environment `https://github.com/SVF-tools/Software-Security-Analysis/wiki/Configure-IDE`
- (2) Write, run and debug your 'hello world' C++ program within your configured IDE.
- (3) Revisit and practice C++ programming (more about programming practices in our lab exercise)
- (4) Start working on Quiz-1 and Lab-1.