CS 453/698: Software and Systems Security

Module: Other Common Vulnerability Types
Lecture: Common mistakes

Meng Xu *(University of Waterloo)*

Spring 2025

Outline

1. Introduction: why study these bug types?
2. Undefined / counterintuitive behaviors
3. Insufficient sanitization on untrusted input
4. Invocation of / by untrusted logic
5. Hostile execution environment
6. Conclusion

"Nice" properties of memory errors

- They have universally accepted definitions
  - Once you find a memory error, 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, 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 in one codebase, you can scale it up to millions of codebases

“Nice” properties of memory errors

- They have universally accepted definitions
  - Once you find a memory error, 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, 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 in one codebase, you can scale it up to millions of codebases

In fact, very few types of vulnerabilities meet these requirements.

“Nice” properties of memory errors

- They have universally accepted definitions
  - Once you find a memory error, 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, 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 in one codebase, you can scale it up to millions of codebases

In fact, very few types of vulnerabilities meet these requirements.
$\implies$ 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 study these bug types?

2. **Undefined / counterintuitive behaviors**

3. Insufficient sanitization on untrusted input

4. Invocation of / by untrusted logic

5. Hostile execution environment

6. Conclusion

Unsafe integer operations

Mathmetical integers are unbounded

WHILE

Machine integers are bounded by a fixed number of bits.

Unsafe integer operations

```
mapping (address => uint256) public balanceOf;

// INSECURE
function transfer(address _to, uint256 _value) {
    /* Check if sender has balance */
    require(balanceOf[msg.sender] >= _value);

    /* Add and subtract new balances */
    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value;
}
```

Q: What is the bug here?

Unsafe integer operations

```
mapping (address => uint256) public balanceOf;

// INSECURE
function transfer(address _to, uint256 _value) {
    /* Check if sender has balance */
    require(balanceOf[msg.sender] >= _value);

    /* Add and subtract new balances */
    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value;
}
```

```
// SECURE
function transfer(address _to, uint256 _value) {
    /* Check if sender has balance and for overflows */
    require(balanceOf[msg.sender] >= _value &&
            balanceOf[_to] + _value >= balanceOf[_to]);

    /* Add and subtract new balances */
    balanceOf[msg.sender] -= _value;
    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?

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?

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?

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;
}
```

Q: Output?

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;
}
```

Q: Output?

```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?

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.

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 compilers will emit a comparison operation based on the numerical value of the pointers.

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 compilers 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 study these bug types?
2. Undefined / counterintuitive behaviors
3. Insufficient sanitization on untrusted input
4. Invocation of / by untrusted logic
5. Hostile execution environment
6. Conclusion

Untrusted input

Handing untrusted input can be dangerous!

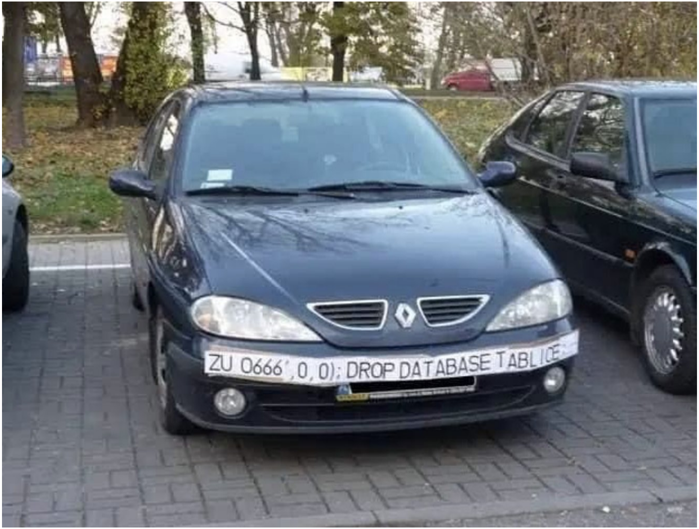
SQL injection

```
public boolean login(String username, String password) {
  String sql =
    "SELECT * FROM Users WHERE " +
      "username = '" + username + "' AND " +
      "password = '" + password + "';";

  ResultSet result = db.executeQuery(sql);
  if (result.next()) {
    /* login success */
    return true;
  } else {
    /* login failure */
    return false;
  }
}
```

Mitigating SQL injection with sanitization

```
public boolean login(String username, String password) {
  PreparedStatement sql = db.prepareStatement(
    "SELECT * FROM Users WHERE username = ? AND password = ?;")
  sql.setString(1, username);
  sql.setString(2, password);

  ResultSet result = db.executeQuery(sql);
  if (result.next()) {
    /* login success */
    return true;
  } else {
    /* login failure */
    return false;
  }
}
```

SQL injection in the wild



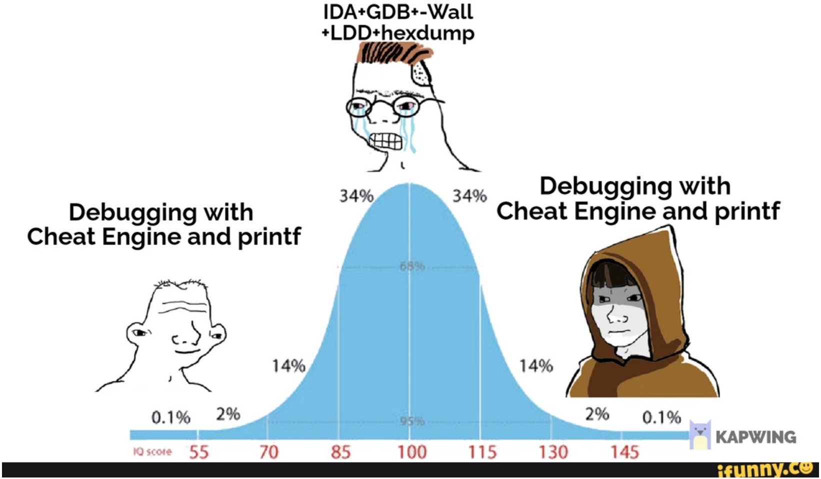
Original source unknown, found on Twitter

`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;
}
```

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 something 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.

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)

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

```python
from urllib.parse import unquote as url_unquote
from http.server import BaseHTTPRequestHandler, HTTPServer

HOST = "localhost"
PORT = 8080

PAGE = """<html>
<form action=’/submit’ method=’POST’>
<input type=’text’ name=’comment’ />
</form>
</html>"""

class XSSDemoServer(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.end_headers()
        self.wfile.write(bytes(PAGE, "utf-8"))

    def do_POST(self):
        size = int(self.headers.get(’Content-Length’))
        body = url_unquote(self.rfile.read(size).decode(’utf-8’))
```

XSS Demo II

```python
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.end_headers()
        self.wfile.write(bytes("<html>%s</html>" % body[8:], "utf-8"))


if __name__ == "__main__":
    server = HTTPServer((HOST, PORT), XSSDemoServer)
    print("Server started http://%s:%s" % (HOST, PORT))

    try:
        server.serve_forever()
    except KeyboardInterrupt:
        pass

    server.server_close()
    print("Server stopped.")
```

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

Outline

1. Introduction: why study these bug types?
2. Undefined / counterintuitive behaviors
3. Insufficient sanitization on untrusted input
4. Invocation of / by untrusted logic
5. Hostile execution environment
6. Conclusion

Calling into untrusted code is dangerous

Calling into untrusted code is dangerous

The DAO attack on Ethereum

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.

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)

```
contract EtherStore {
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(balances[msg.sender] >= _weiToWithdraw);
        require(_weiToWithdraw <= withdrawalLimit);
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());

        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
 }
```

Reentrancy attack (attacker's contract)

```
import "EtherStore.sol";

contract Attack {
  EtherStore public etherStore;

  constructor(address _etherStoreAddress) {
      etherStore = EtherStore(_etherStoreAddress);
  }
  function pwnEtherStore() public payable {
      require(msg.value >= 1 ether);
      etherStore.depositFunds.value(1 ether)();
      etherStore.withdrawFunds(1 ether);
  }
  function collectEther() public {
      msg.sender.transfer(this.balance);
  }
  function () payable {
      if (etherStore.balance > 1 ether) {
          etherStore.withdrawFunds(1 ether);
      }
  }
}
```

Reentrancy attack (attacker's contract)

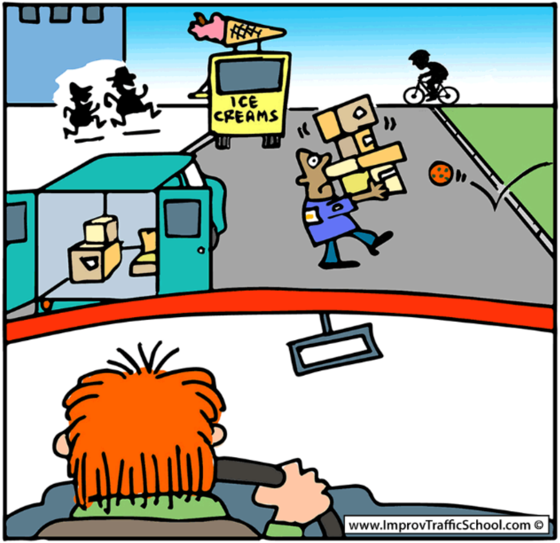
```
import "EtherStore.sol";

contract Attack {
  EtherStore public etherStore;

  constructor(address _etherStoreAddress) {
      etherStore = EtherStore(_etherStoreAddress);
  }
  function pwnEtherStore() public payable {
      require(msg.value >= 1 ether);
      etherStore.depositFunds.value(1 ether)();
      etherStore.withdrawFunds(1 ether);
  }
  function collectEther() public {
      msg.sender.transfer(this.balance);
  }
  function () payable {
      if (etherStore.balance > 1 ether) {
          etherStore.withdrawFunds(1 ether);
      }
  }
}
```

The attacker can drain all balance from the victim contract.

Reentrancy attack (the fix)

```
contract EtherStore {
    bool reentrancyMutex = false;
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(balances[msg.sender] >= _weiToWithdraw);
        require(_weiToWithdraw <= withdrawalLimit);
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);

        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
        reentrancyMutex = true;
        msg.sender.transfer(_weiToWithdraw);
        reentrancyMutex = false;
    }
 }
```

Defensive driving



Credits / Trademark: `www.ImprovTafficSchool.com`

Defensive programming

Like defensive driving, defensive programming requires the developer to anticipate what might go wrong in the software and program defensively against these anticipated issues, potentially with the help of compiler, runtime, or even external auditors.

Defensive programming

Driving	**Programming**
Follow traffic rules Follow local customs	Follow typing rules Follow coding conventions

In normal paradigm: expect others to follow the rules
In defensive paradigm: expect others to ignore / by-pass the rules

Defensive programming

Driving	**Programming**
Follow traffic rules Follow local customs	Follow typing rules Follow coding conventions

In normal paradigm: expect others to follow the rules
In defensive paradigm: expect others to ignore / by-pass the rules

Apply defensive actions **at the cost of performance**

Outline

1. Introduction: why study these bug types?
2. Undefined / counterintuitive behaviors
3. Insufficient sanitization on untrusted input
4. Invocation of / by untrusted logic
5. Hostile execution environment
6. Conclusion

Hostile environment

Sometimes the execution environment cannot be trusted as well!

Front-running

```
contract FindThisHash {
  // the keccak-256 hash of some secret string
  bytes32 constant public hash
    = 0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;

  constructor() public payable {} // load with ether

  function solve(string solution) public {
    // If you can find the pre image of the hash, receive 1000 ether
    require(hash == sha3(solution));
    msg.sender.transfer(1000 ether);
  }
}
```

Q: What is the secret string?

Front-running

```
contract FindThisHash {
  // the keccak-256 hash of some secret string
  bytes32 constant public hash
    = 0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;

  constructor() public payable {} // load with ether

  function solve(string solution) public {
    // If you can find the pre image of the hash, receive 1000 ether
    require(hash == sha3(solution));
    msg.sender.transfer(1000 ether);
  }
}
```

Q: What is the secret string?

A: Ethereum!

Front-running

```
contract FindThisHash {
  // the keccak-256 hash of some secret string
  bytes32 constant public hash
    = 0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;

  constructor() public payable {} // load with ether

  function solve(string solution) public {
    // If you can find the pre image of the hash, receive 1000 ether
    require(hash == sha3(solution));
    msg.sender.transfer(1000 ether);
  }
}
```

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.

Sandwich attack

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

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.

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.

Block timestamp dependence

```
contract Roulette {
    uint public pastBlockTime; // Forces one bet per block

    constructor() public payable {} // initially fund contract

    // fallback function used to make a bet
    function () public payable {
        require(msg.value == 10 ether); // must send 10 ether to play
        require(now != pastBlockTime); // only 1 transaction per block
        pastBlockTime = now;
        if(now % 15 == 0) { // winner
            msg.sender.transfer(this.balance);
        }
    }
}
```

Block timestamp dependence

```
contract Roulette {
    uint public pastBlockTime; // Forces one bet per block

    constructor() public payable {} // initially fund contract

    // fallback function used to make a bet
    function () public payable {
        require(msg.value == 10 ether); // must send 10 ether to play
        require(now != pastBlockTime); // only 1 transaction per block
        pastBlockTime = now;
        if(now % 15 == 0) { // winner
            msg.sender.transfer(this.balance);
        }
    }
}
```

The 15-second rule: On Ethereum, a miner can post a timestamp within 15 seconds of the block being validated. This effectively allows the miner to pre-compute an option more favorable to its chances in the lottery — timestamps are not truly random!

Replay attacks

```
function transferProxy(
    address _from, address _to, uint256 _value, uint256 _fee,
    uint8 _v, bytes32 _r, bytes32 _s
) public returns (bool) {
    if (balances[_from] < _fee + _value || _fee > _fee + _value) revert();

    uint256 nonce = nonces[_from];
    bytes32 h = keccak256(_from,_to,_value,_fee,nonce);
    if (_from != ecrecover(h,_v,_r,_s)) revert();

    if (balances[_to] + _value < balances[_to]
        || balances[msg.sender] + _fee < balances[msg.sender]) revert();
    balances[_to] += _value;
    emit Transfer(_from, _to, _value);

    balances[msg.sender] += _fee;
    emit Transfer(_from, msg.sender, _fee);

    balances[_from] -= _value + _fee;
    nonces[_from] = nonce + 1;
    return true;
}
```

Replay attacks

```
function transferProxy(
    address _from, address _to, uint256 _value, uint256 _fee,
    uint8 _v, bytes32 _r, bytes32 _s
) public returns (bool) {
    if (balances[_from] < _fee + _value || _fee > _fee + _value) revert();

    uint256 nonce = nonces[_from];
    bytes32 h = keccak256(_from,_to,_value,_fee,nonce);
    if (_from != ecrecover(h,_v,_r,_s)) revert();

    if (balances[_to] + _value < balances[_to]
        || balances[msg.sender] + _fee < balances[msg.sender]) revert();
    balances[_to] += _value;
    emit Transfer(_from, _to, _value);

    balances[msg.sender] += _fee;
    emit Transfer(_from, msg.sender, _fee);

    balances[_from] -= _value + _fee;
    nonces[_from] = nonce + 1;
    return true;
}
```

This function can be replayed with another token!

Outline

1. Introduction: why study these bug types?
2. Undefined / counterintuitive behaviors
3. Insufficient sanitization on untrusted input
4. Invocation of / by untrusted logic
5. Hostile execution environment
6. Conclusion

Conclusion

- Don't make assumptions about the programming language
- Don't make assumptions about user inputs
- Don't make assumptions about code you call into or code that calls into your program
- Don't make assumptions about the execution environment

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