CMPT 473
Software Testing, Reliability and Security

Security

Nick Sumner

Security in General

- ***Security***
  - Maintaining desired properties in the the presence of adversaries

Security in General

- ***Security***
  - Maintaining desired properties in the the presence of adversaries

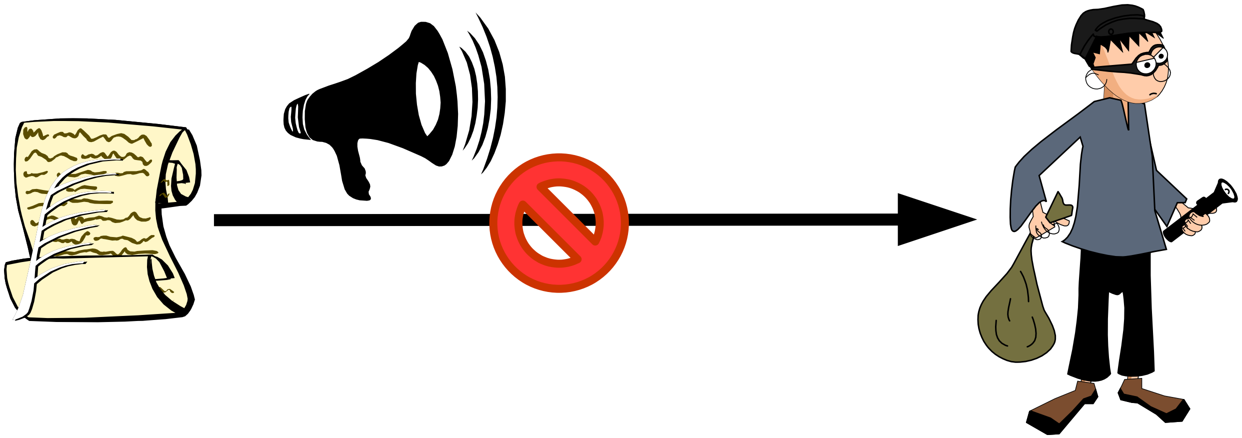
So what are the desired properties?

Security in General

- ***Security***
  - Maintaining desired properties in the the presence of adversaries
- CIA Model – classic security properties

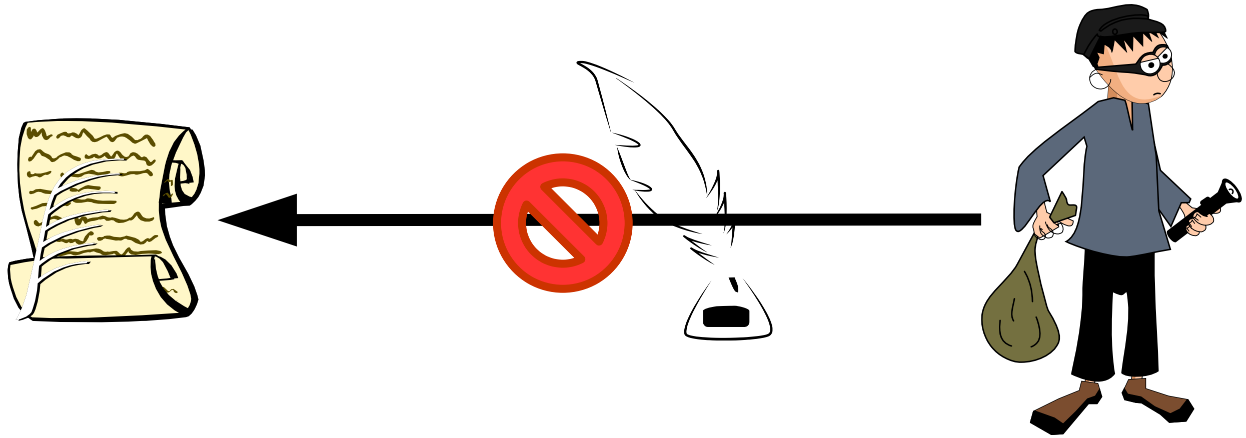
Security in General

- ***Security***
  - Maintaining desired properties in the the presence of adversaries
- CIA Model – classic security properties
  - **Confidentiality**
    - Information is only disclosed to those authorized to know it



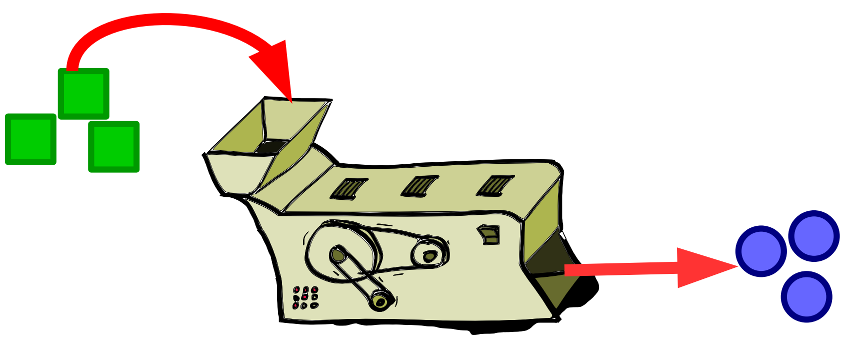
Security in General

- ***Security***
  - Maintaining desired properties in the the presence of adversaries
- CIA Model – classic security properties
  - Confidentiality
  - **Integrity**
    - Only modify information in allowed ways by authorized parties



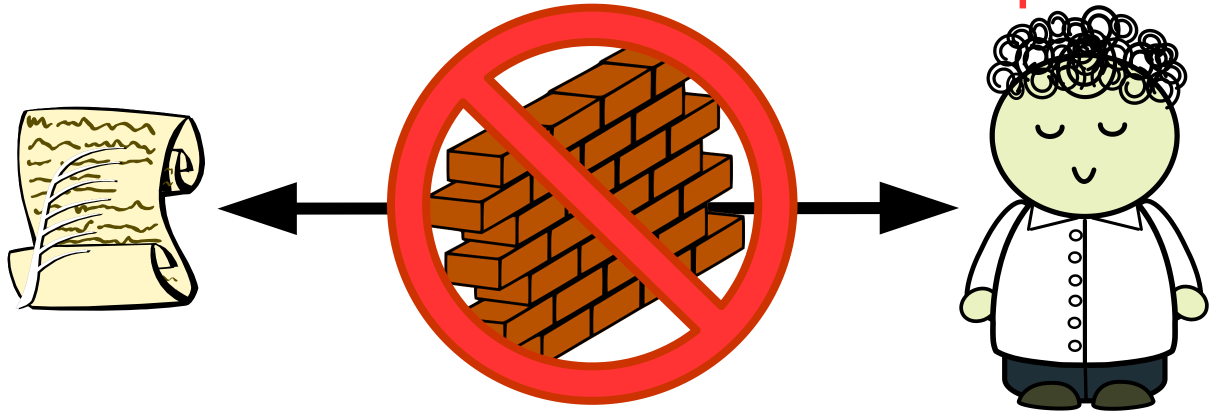
Security in General

- ***Security***
  - Maintaining desired properties in the the presence of adversaries
- CIA Model – classic security properties
  - Confidentiality
  - **Integrity**
    - Only modify information in allowed ways by authorized parties
    - Do what is expected



Security in General

- *Security*
  - Maintaining desired properties in the the presence of adversaries
- CIA Model – classic security properties
  - Confidentiality
  - Integrity
  - **Availability**
    - Those authorized for access are not prevented from it



Security in Software

- *Bugs* in software can lead to policy violations
  - Information leaks (**C**)

Security in Software

- *Bugs* in software can lead to policy violations
  - Information leaks (**C**)
  - Data Corruption (**I**)

Security in Software

- *Bugs* in software can lead to policy violations
  - Information leaks (**C**)
  - Data Corruption (**I**)
  - Denial of service (**A**)

Security in Software

- *Bugs* in software can lead to policy violations
  - Information leaks (**C**)
  - Data Corruption (**I**)
  - Denial of service (**A**)
  - Remote execution – (**CIA**) arbitrarily bad!

Security in Software

- *Bugs* in software can lead to policy violations
- *Bugs* make software vulnerable to attack

Security in Software

- *Bugs* in software can lead to policy violations
- *Bugs* make software vulnerable to attack
  - XSS
  - SQL Injection
  - Buffer overflow
  - Path replacement
  - Integer overflow
  - Race conditions (TOCTOU – Time of Check to Time of Use)
  - Unsanitized format strings
  - …

All create attack vectors for a malicious adversary

Why Is This Special?

Poor security comes from unintended behavior.

→ Quality software shouldn't allow such actions anyway.

Why Is This Special?

Poor security comes from unintended behavior.

→ Quality software shouldn't allow such actions anyway.

- While our testing techniques so far find some security issues, many slip through! *Why?*

Why Is This Special?

Poor security comes from unintended behavior.

→ Quality software shouldn't allow such actions anyway.

- While our testing techniques so far find some security issues, many slip through! *Why?*
  - We cannot test everything

Why Is This Special?

Poor security comes from unintended behavior.

→ Quality software shouldn't allow such actions anyway.

- While our testing techniques so far find some security issues, many slip through! *Why?*
  - We cannot test everything
  - Concessions form part of an *attack surface*
    - Networks, Software, People

Why Is This Special?

Poor security comes from unintended behavior.

→ Quality software shouldn't allow such actions anyway.

- While our testing techniques so far find some security issues, many slip through! *Why?*
  - We cannot test everything
  - Concessions form part of an *attack surface*
    - Networks, Software, People
- Need additional policies & testing methods that specifically address security

What Could Possible Go Wrong?

- Many ways to attack different programs
- MITRE groups the most common into:

What Could Possible Go Wrong?

- Many ways to attack different programs
- MITRE groups the most common into:
  - Insecure Interaction
    - Data sent between components in an insecure fashion

What Could Possible Go Wrong?

- Many ways to attack different programs
- MITRE groups the most common into:
  - Insecure Interaction
    - Data sent between components in an insecure fashion
  - Risky Resource Management
    - Bad creation, use, transfer, & destruction of resources

What Could Possible Go Wrong?

- Many ways to attack different programs
- MITRE groups the most common into:
  - Insecure Interaction
    - Data sent between components in an insecure fashion
  - Risky Resource Management
    - Bad creation, use, transfer, & destruction of resources
  - Porous Defenses
    - Standard security practices that are missing or incorrect

[http://cwe.mitre.org/top25/#Categories]

Memory Safety

- *Unsafe memory* accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]

Memory Safety

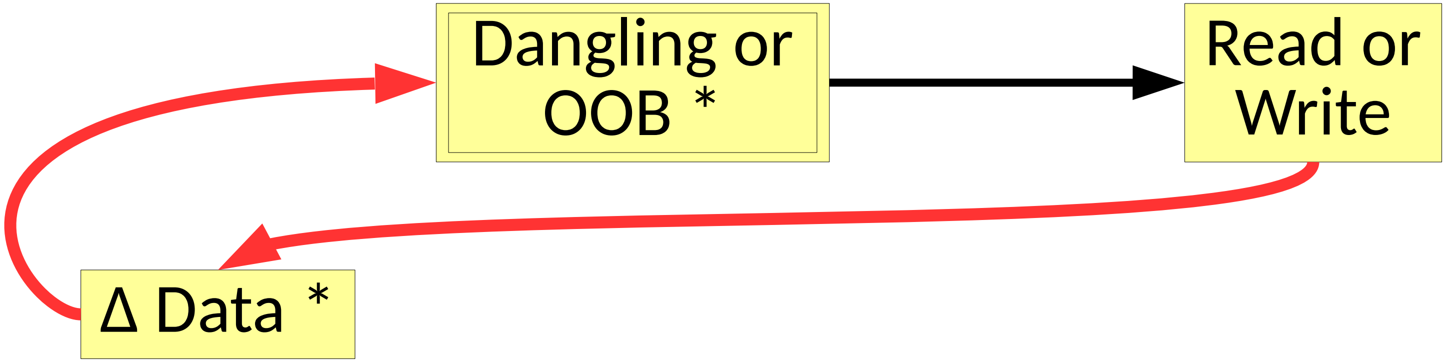
- *Unsafe memory* accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]
- Provide common attack patterns [Eternal War in Memory]

Memory Safety

- *Unsafe memory* accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]
- Provide common attack patterns [Eternal War in Memory]

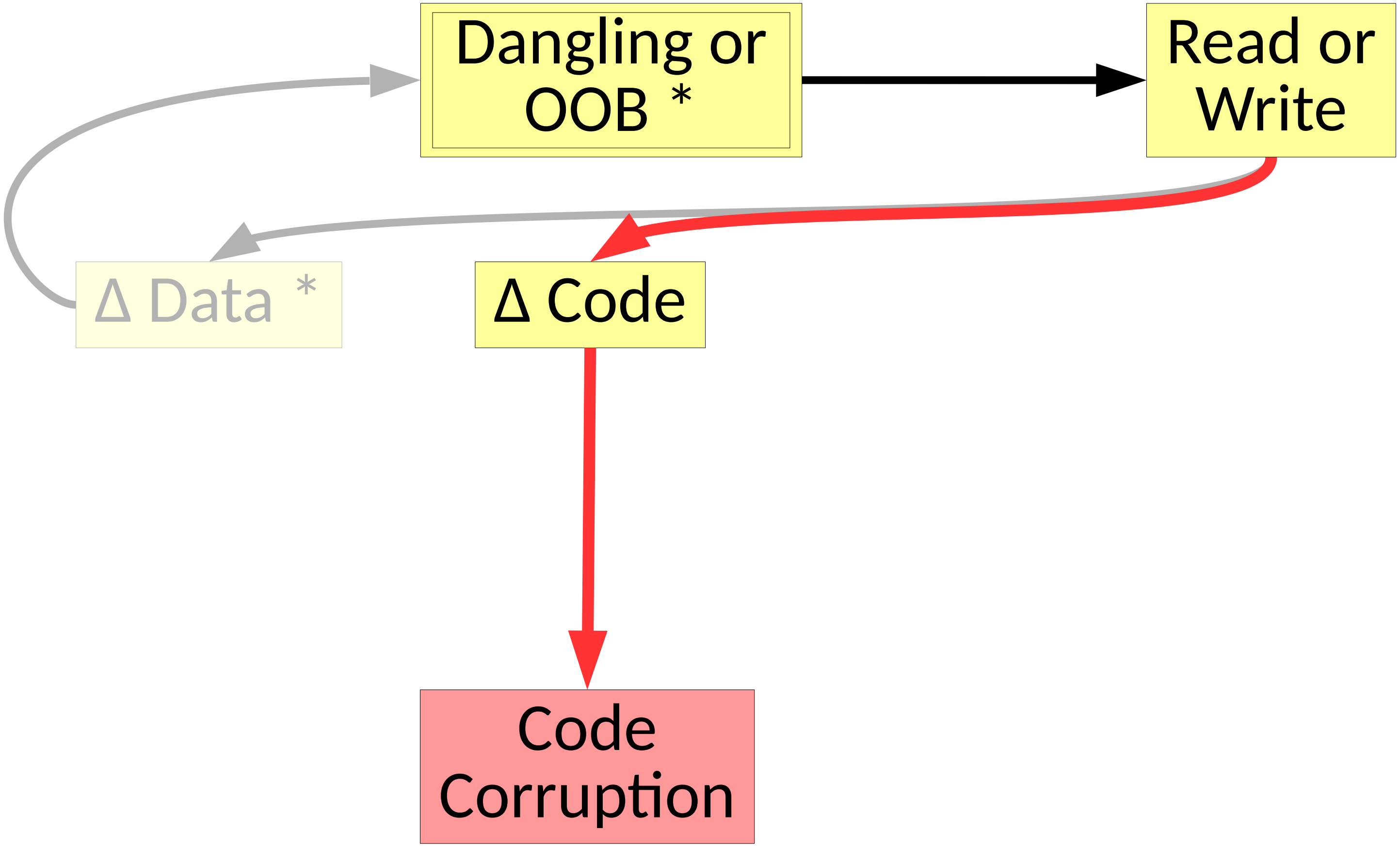
Memory Safety

- *Unsafe memory* accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]
- Provide common attack patterns [Eternal War in Memory]



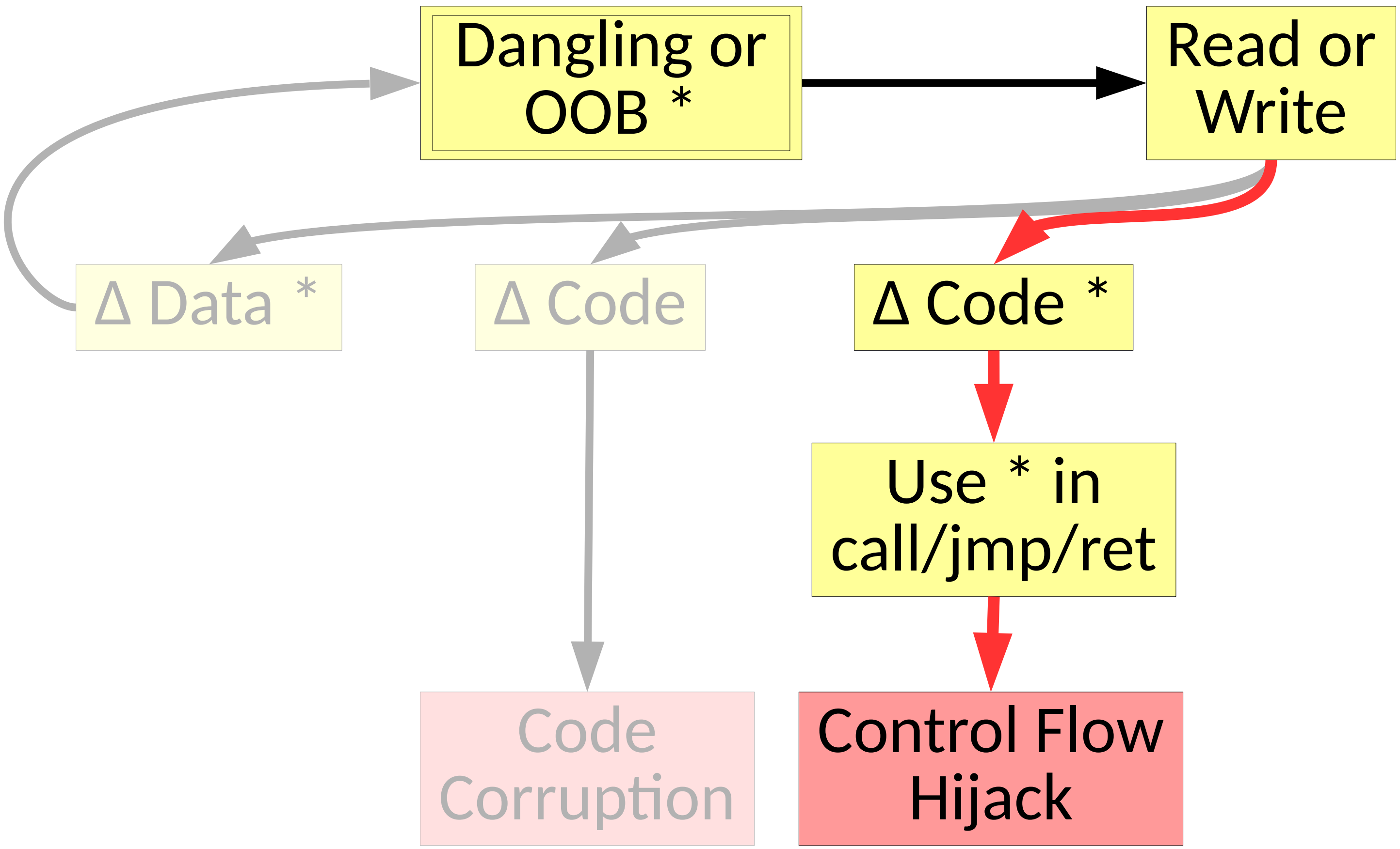
Memory Safety

- *Unsafe memory* accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]
- Provide common attack patterns [Eternal War in Memory]



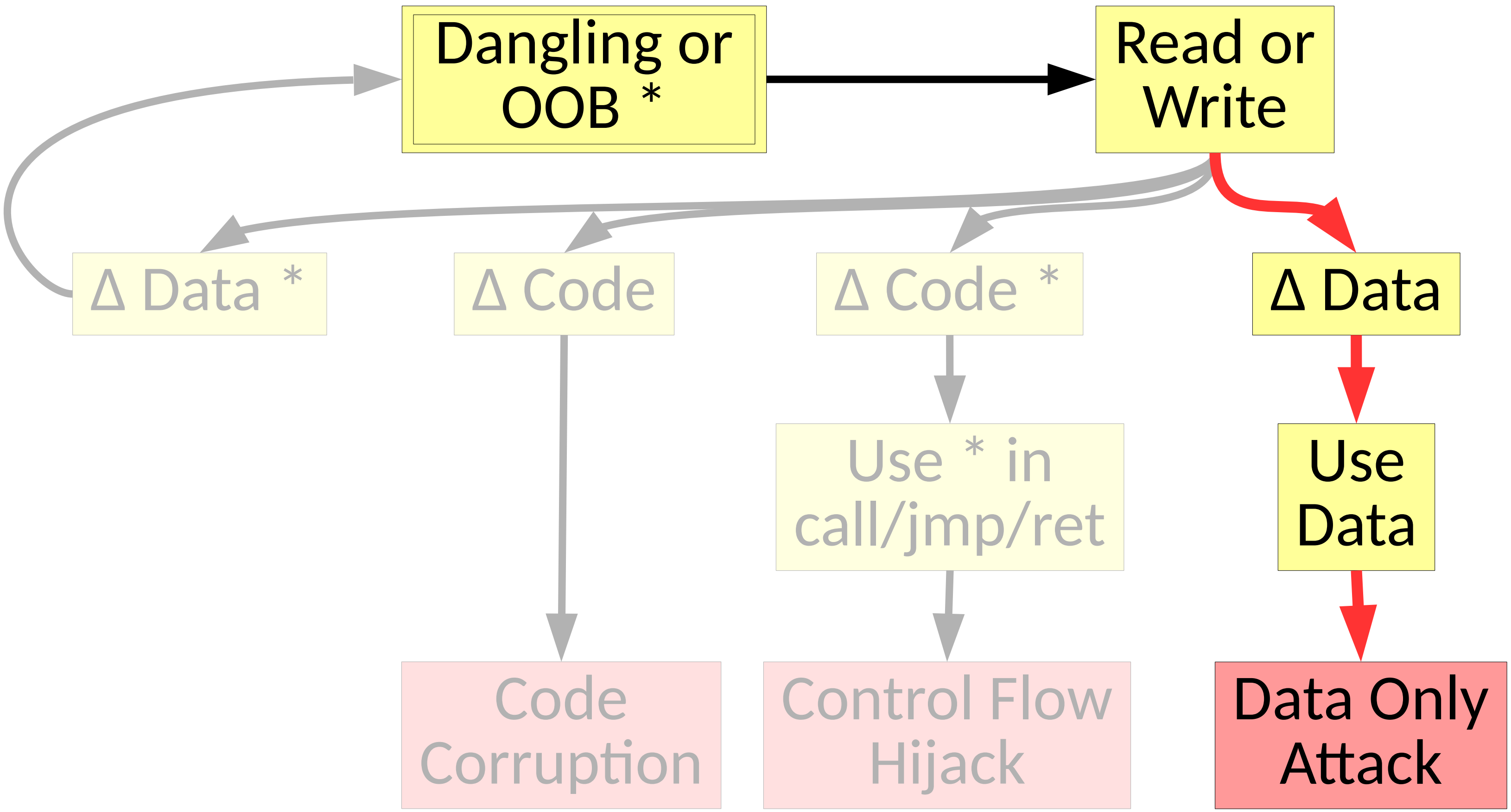
Memory Safety

- *Unsafe memory* accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]
- Provide common attack patterns [Eternal War in Memory]



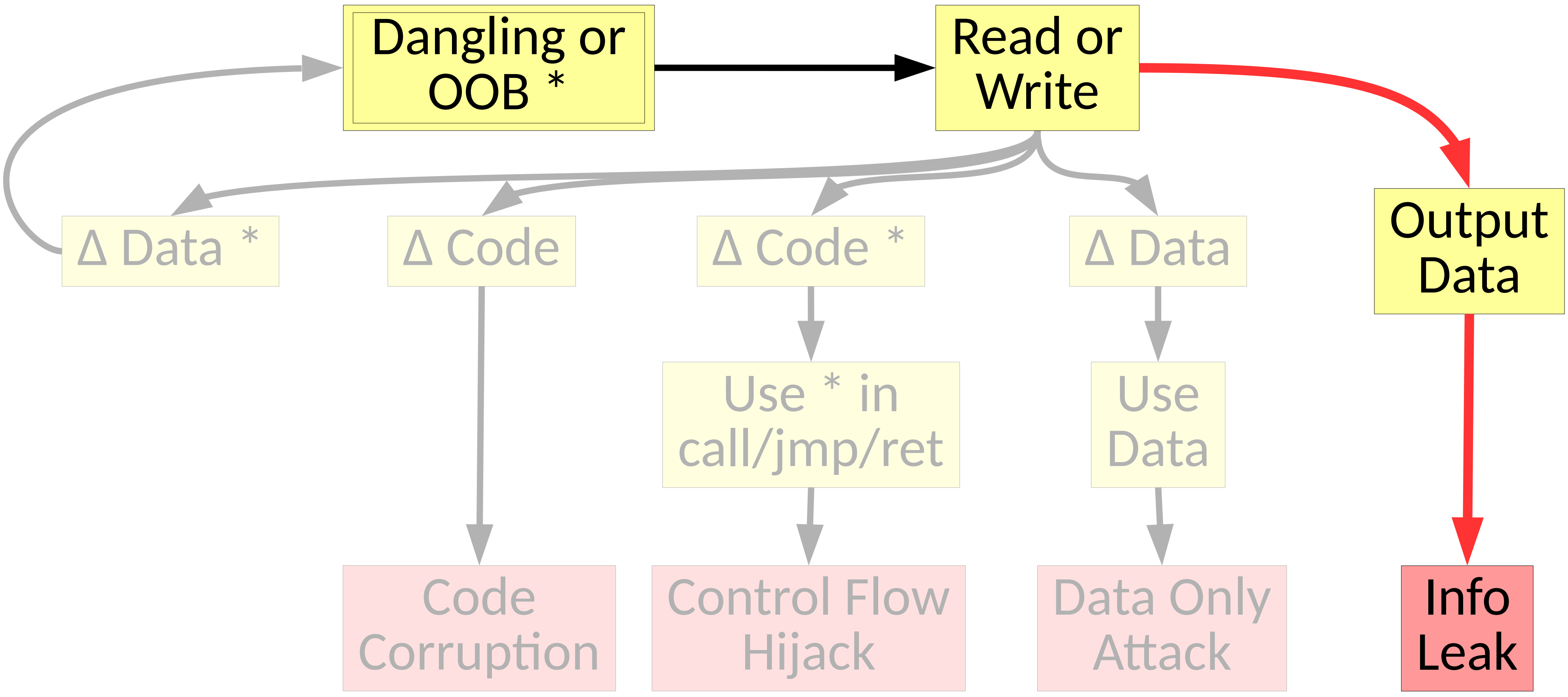
Memory Safety

- *Unsafe memory* accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]
- Provide common attack patterns [Eternal War in Memory]



Memory Safety

- *Unsafe memory* accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]
- Provide common attack patterns [Eternal War in Memory]



Code Corruption

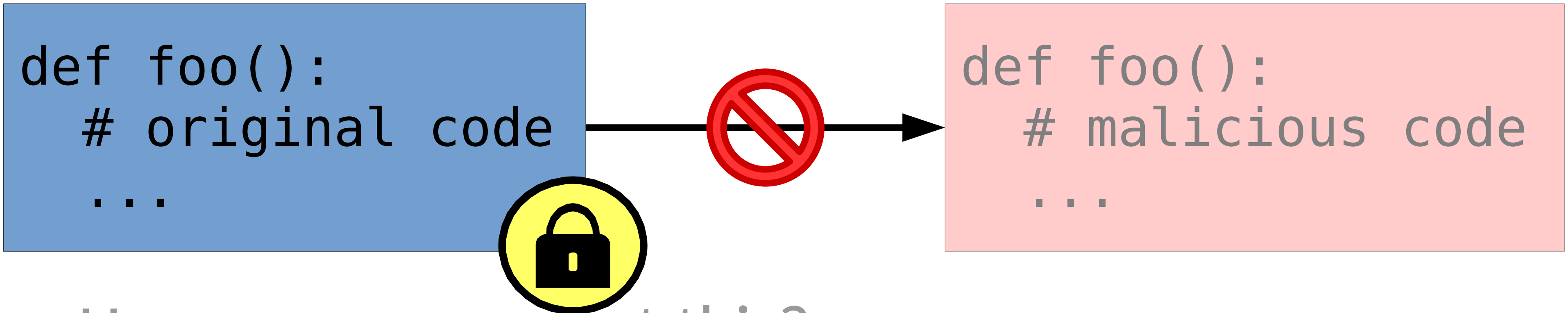
```
def foo():
  # original code
  ...
```

→

```
def foo():
  # malicious code
  ...
```

- How can we prevent this?

Code Corruption



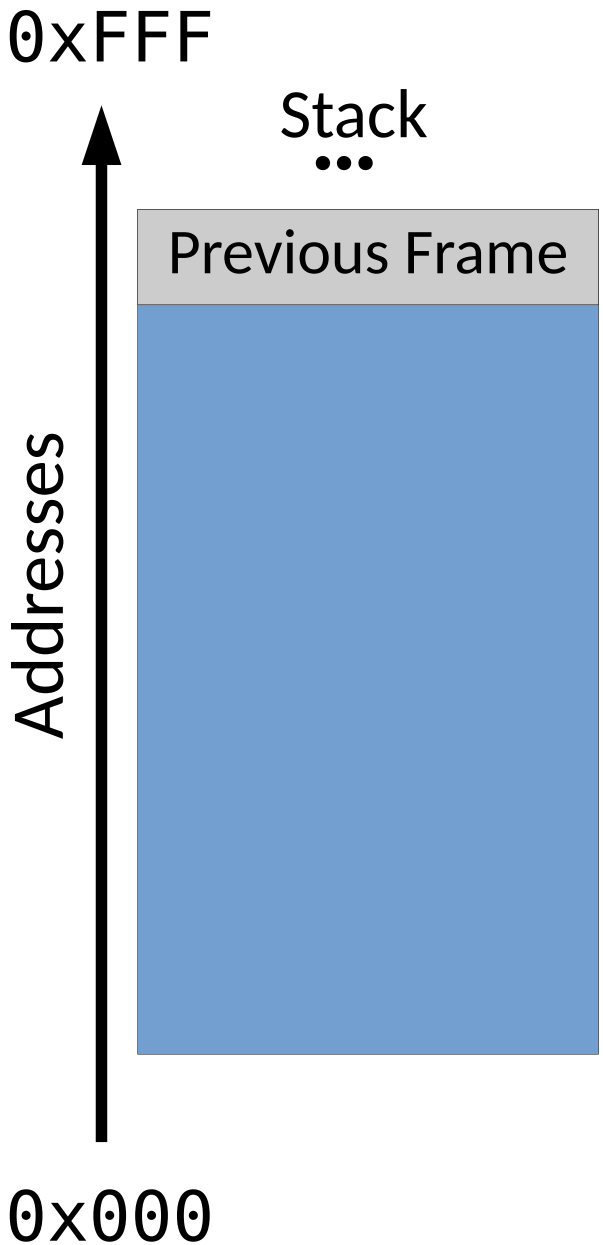
- How can we prevent this?
- What problems does this solution create?

Control Flow Hijacking

```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

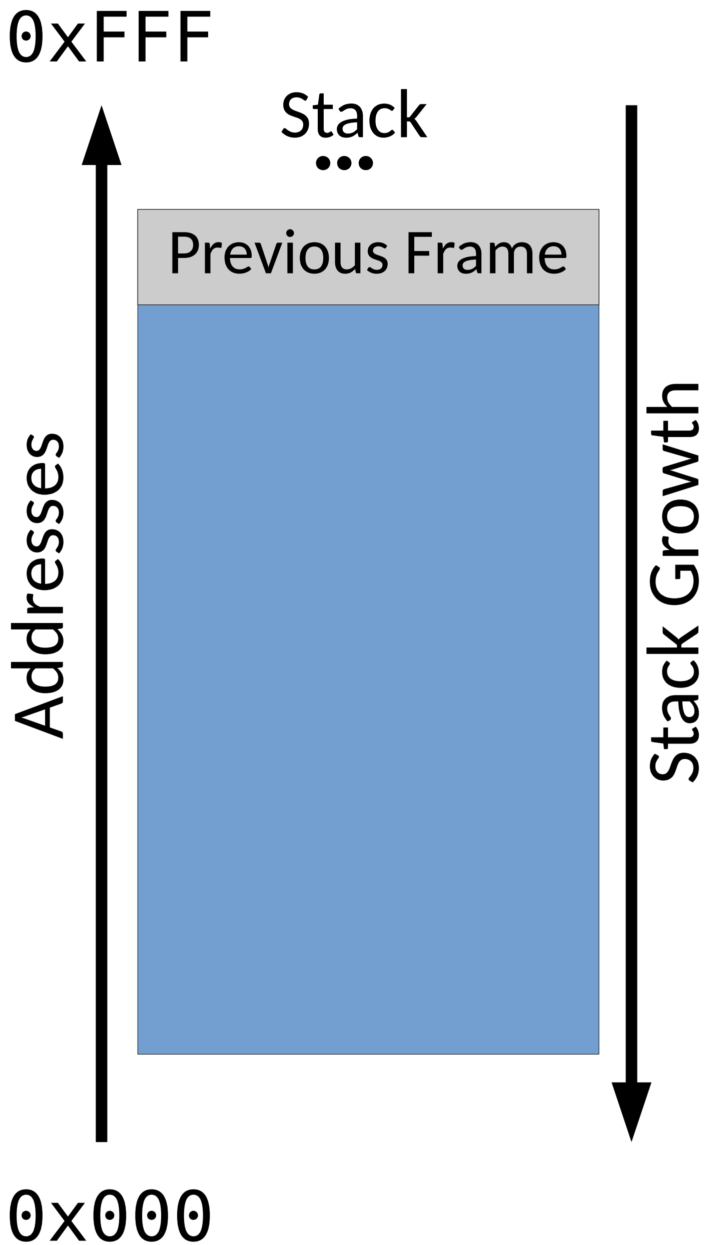
How many of you recall what a stack frame looks like?

Data Only Attacks



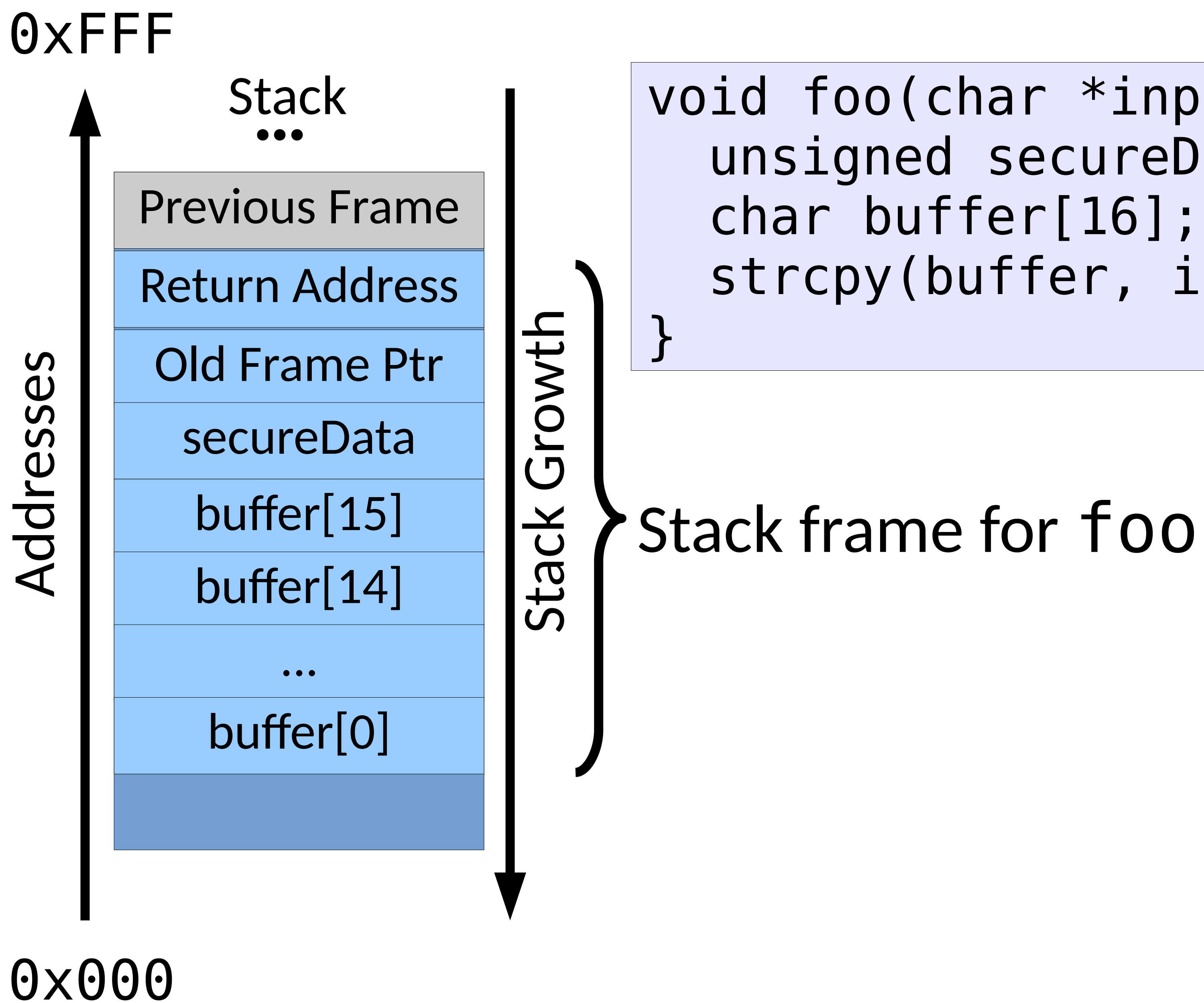

```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

Data Only Attacks



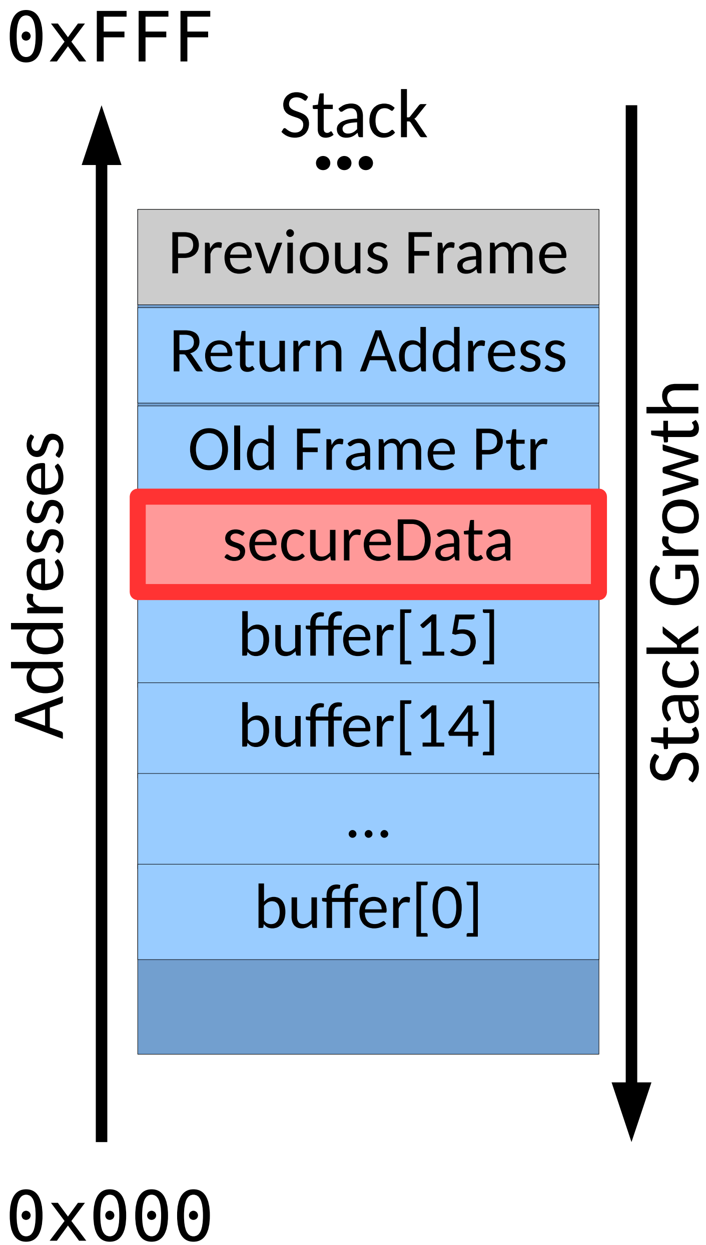

```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

Data Only Attacks



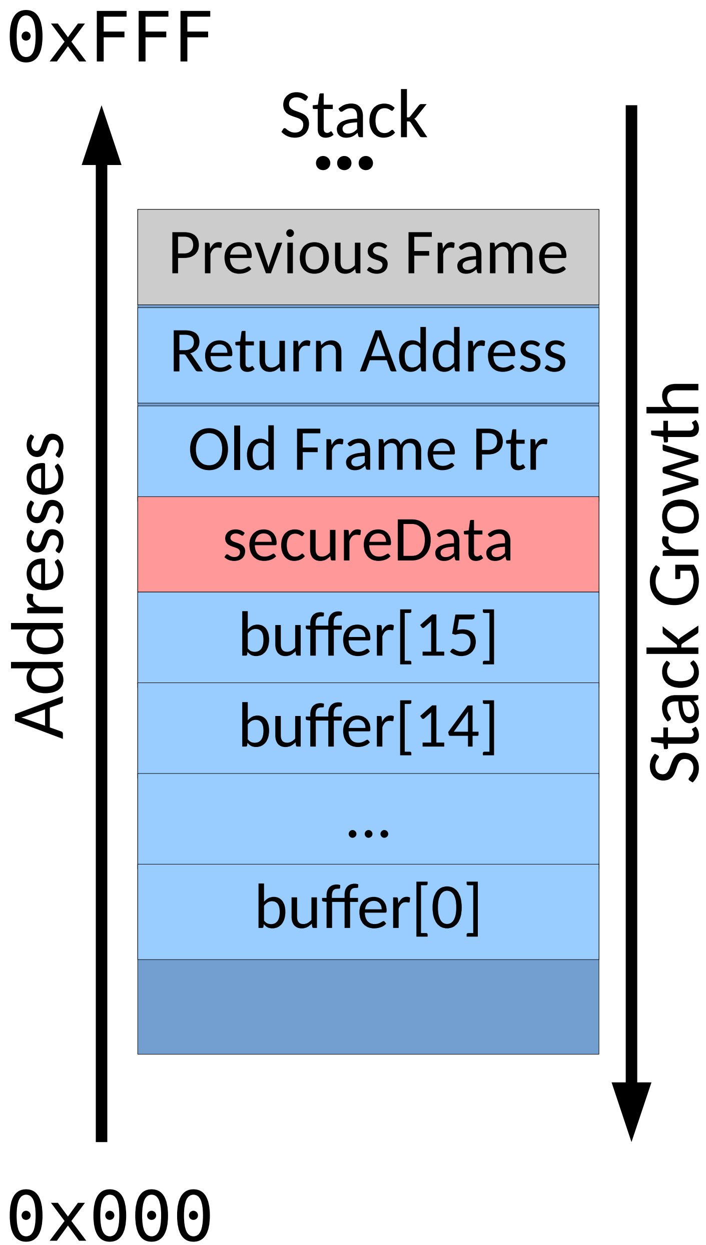

```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

Data Only Attacks




```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

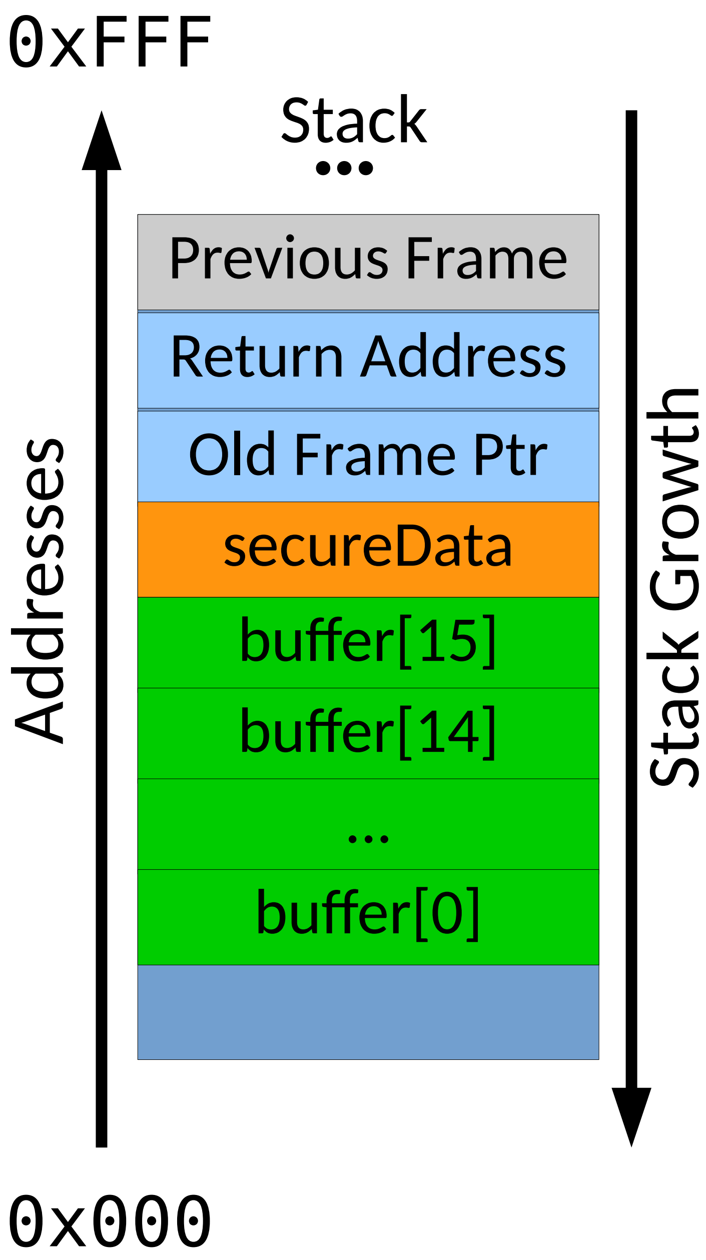
Data Only Attacks




```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

What can go wrong?

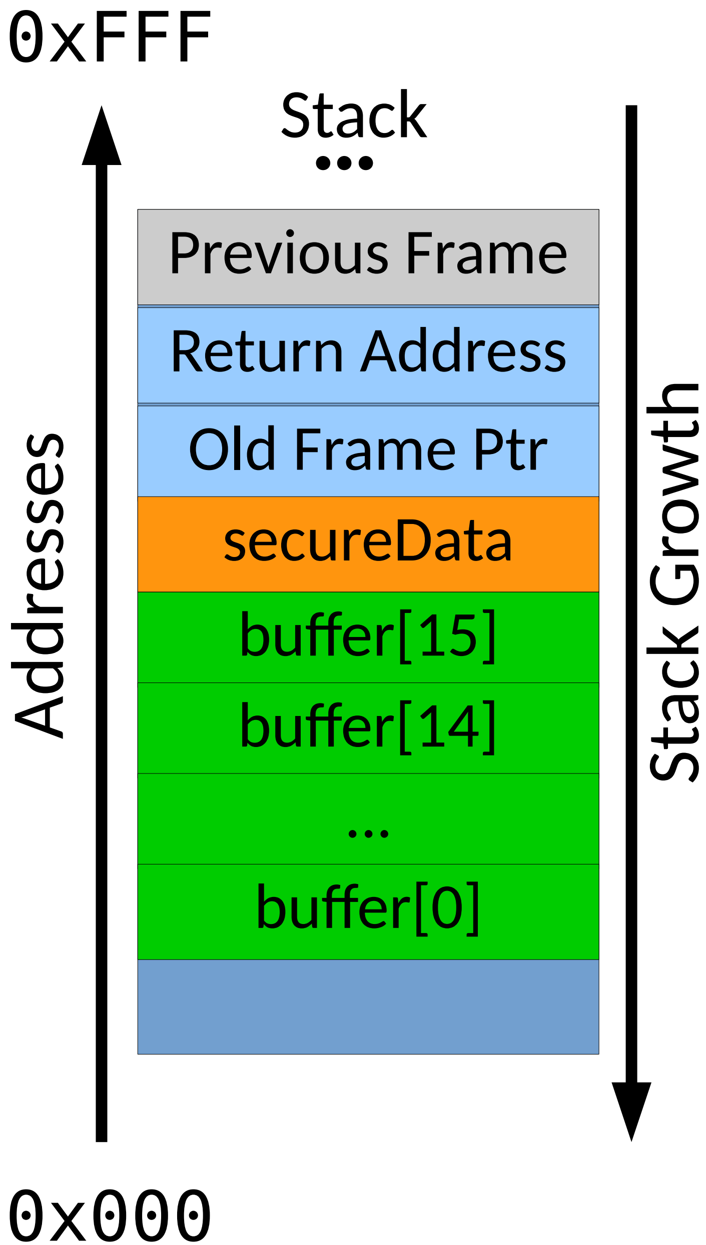
Data Only Attacks




```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

input = “normal input”
+ “insecureData”

Data Only Attacks

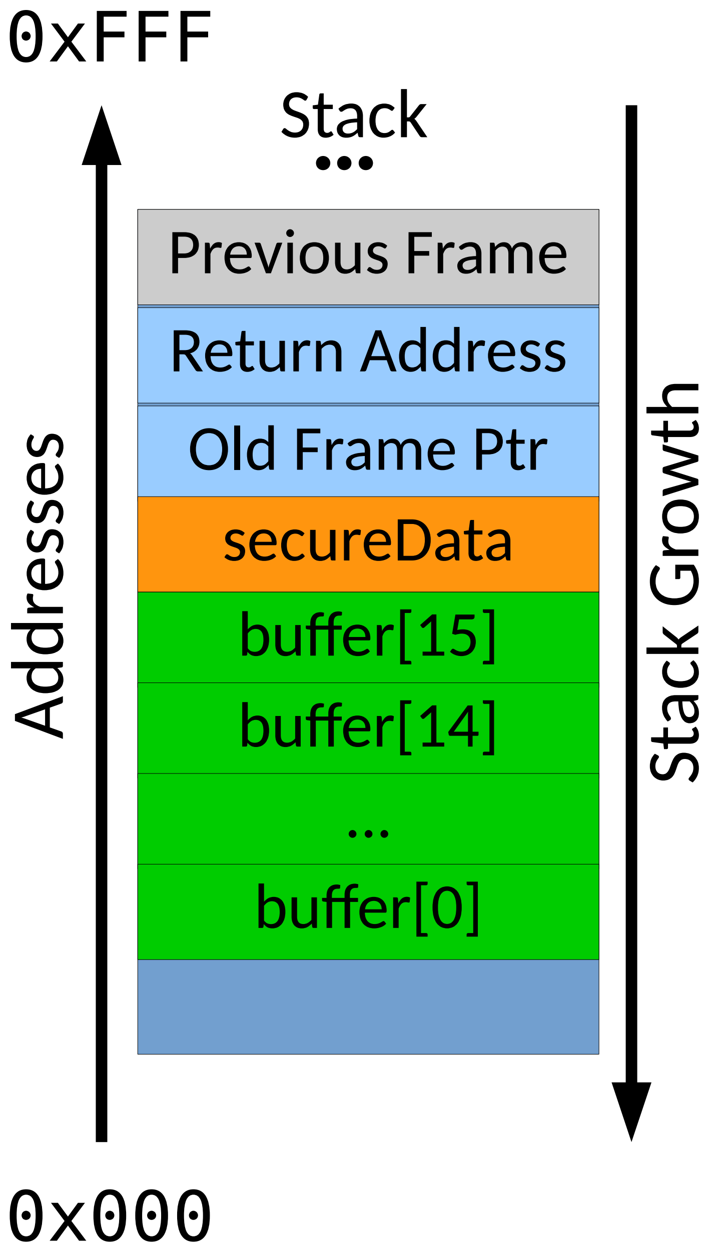



```c
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

input = "normal input"
+ "insecureData"

buffer overflow attack

Data Only Attacks

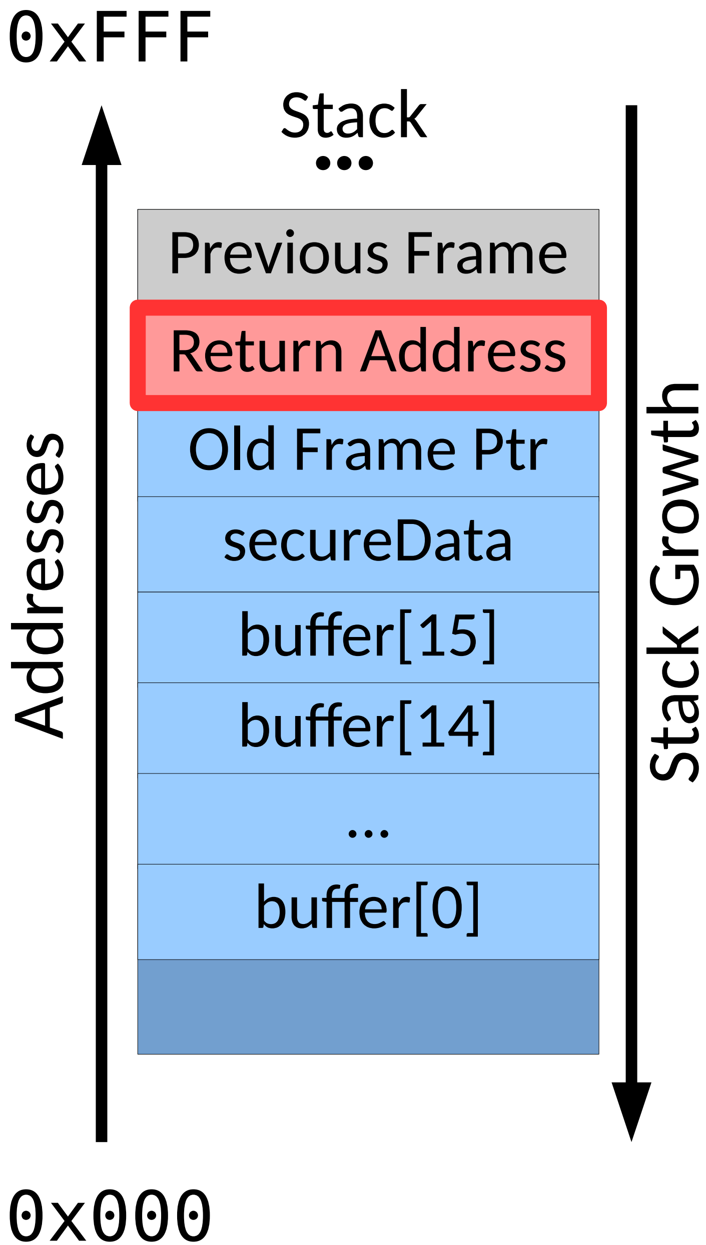



```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

input = "normal input"
+ "insecureData"

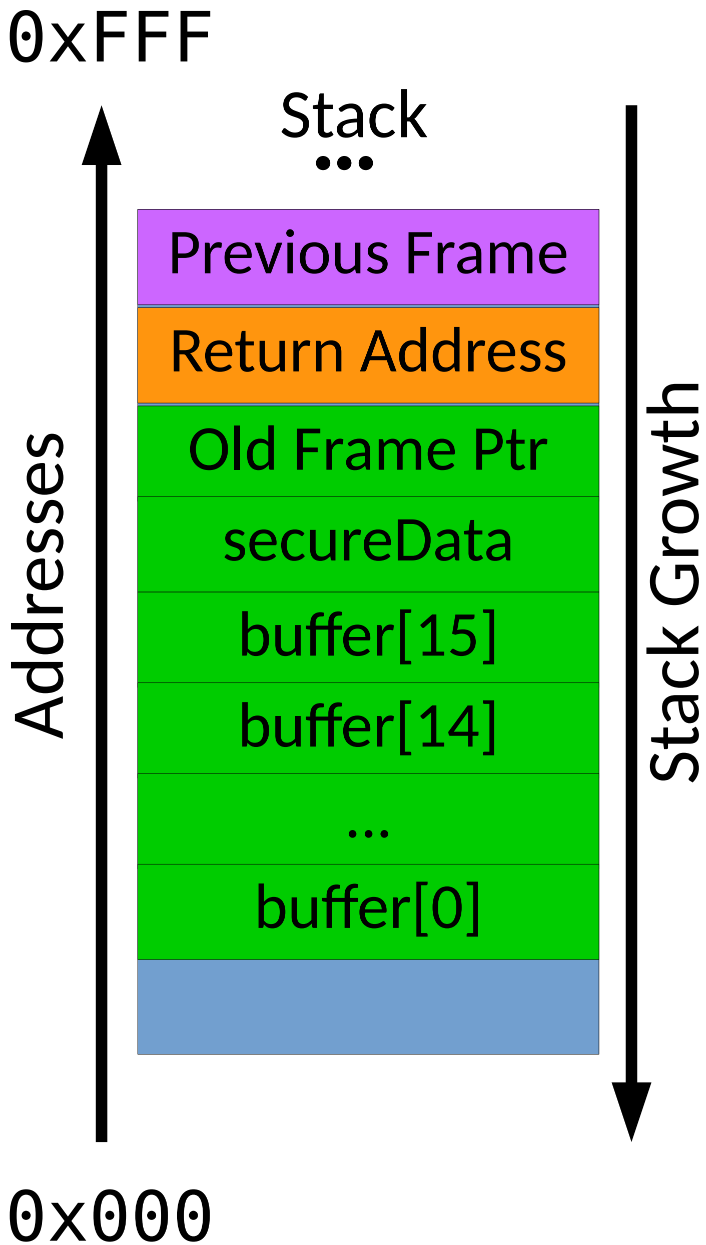
The integrity of the secure data is corrupted.

Control Flow Hijacking




```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

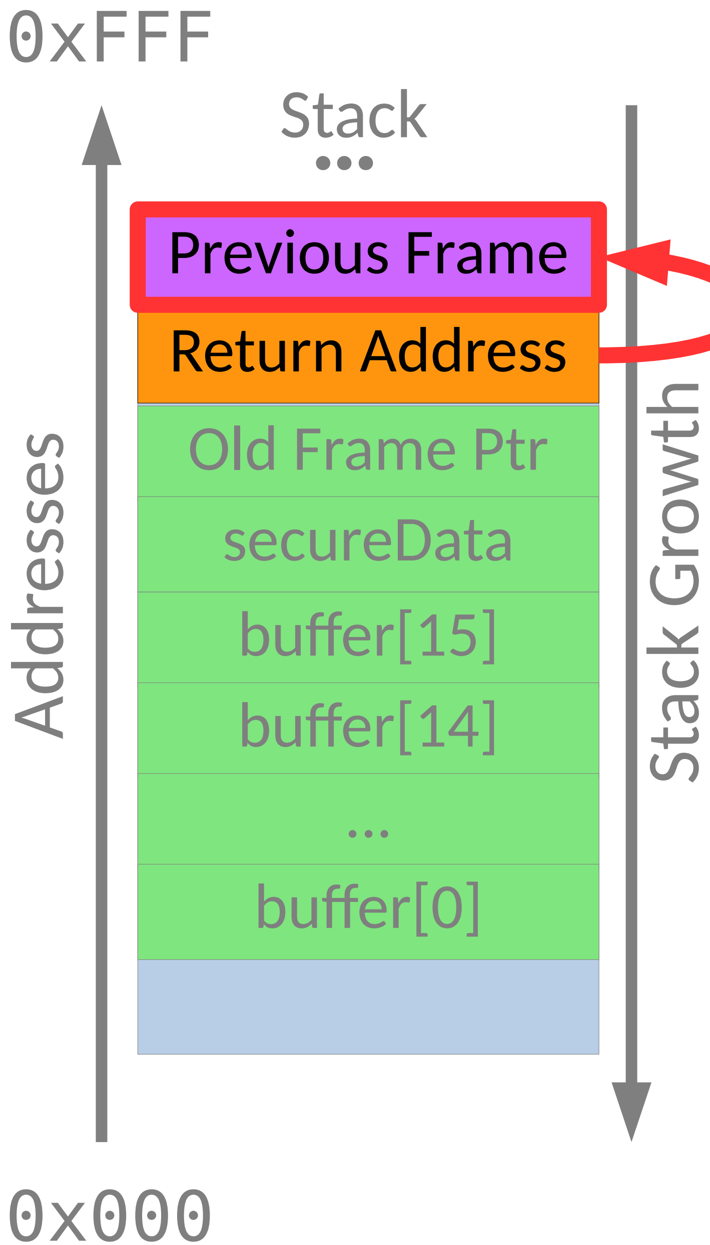
Control Flow Hijacking




```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

input = “input”
+ “payload address”
+ “payload (shell code)”

Control Flow Hijacking




```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

input = “input”
+ “payload address”
+ “payload (shell code)”

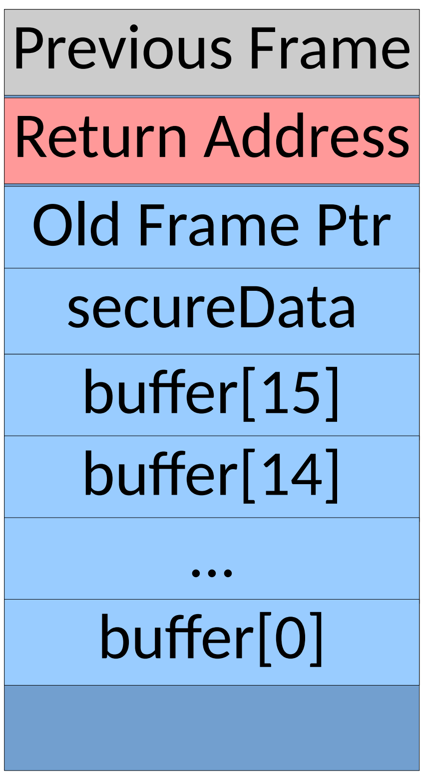
On return, we'll execute the shell code

Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries

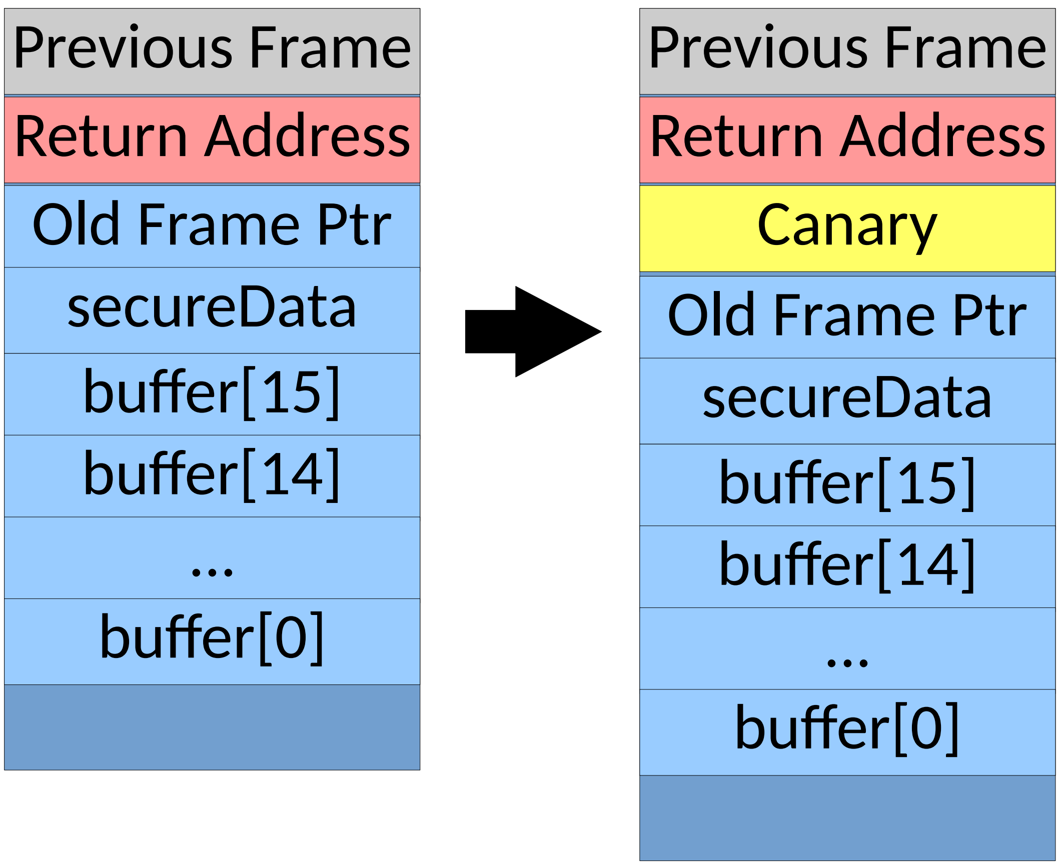
Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries



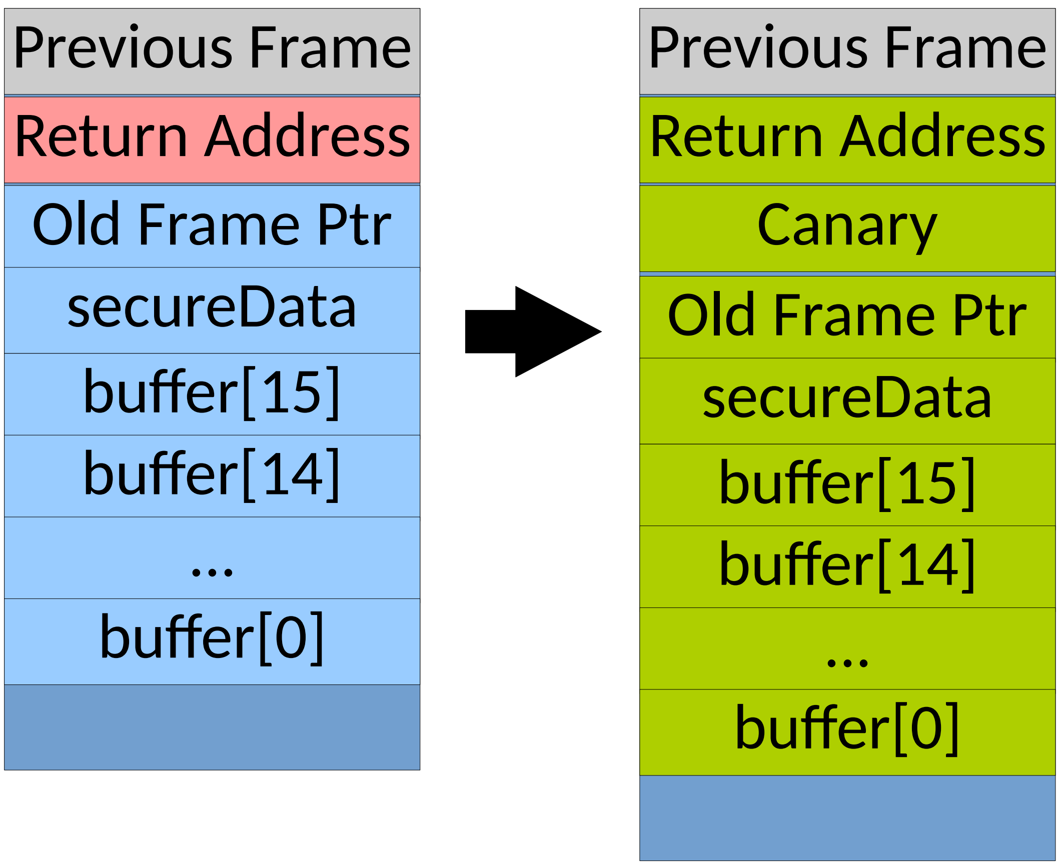
Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries



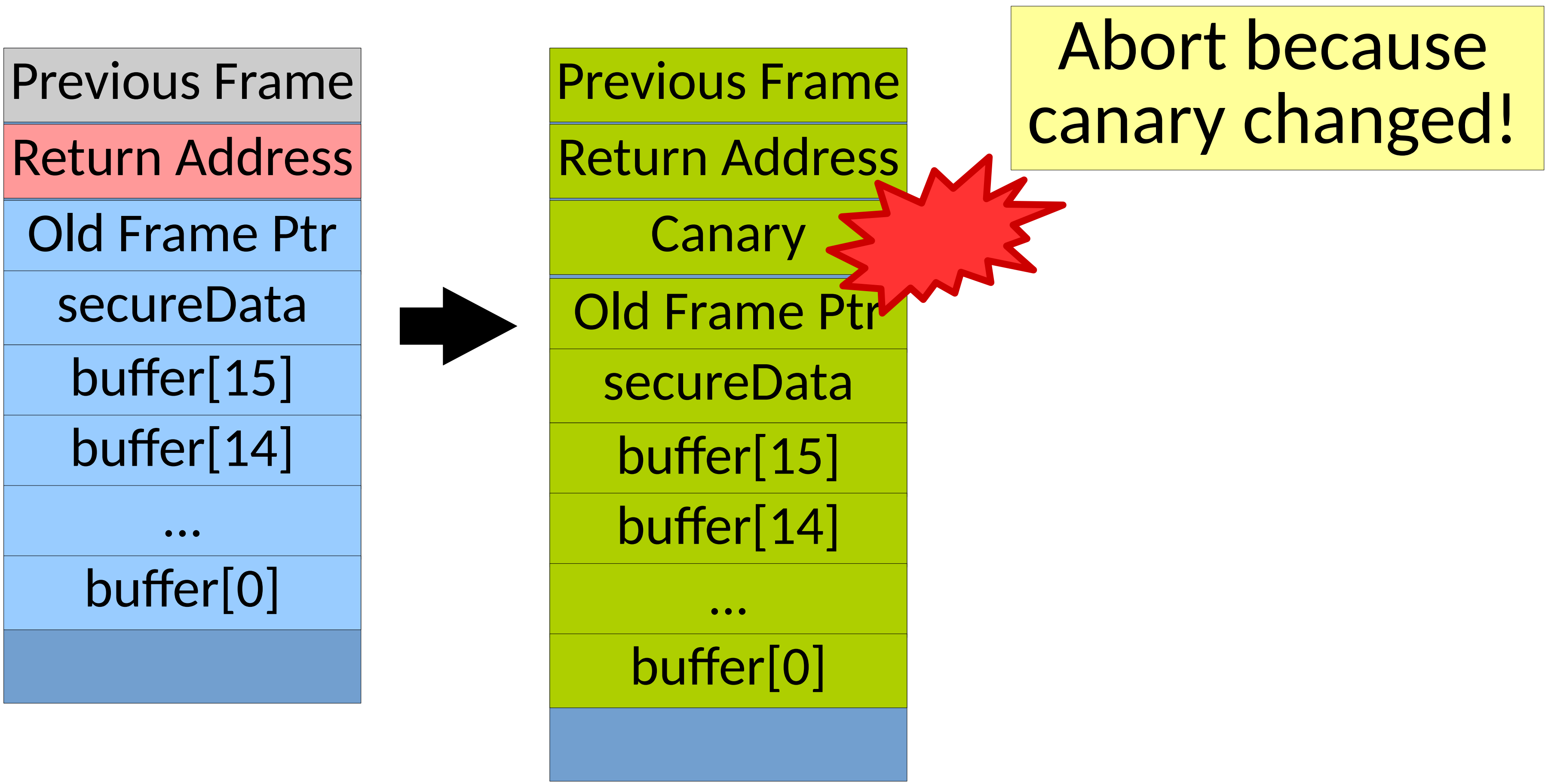
Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries



Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries

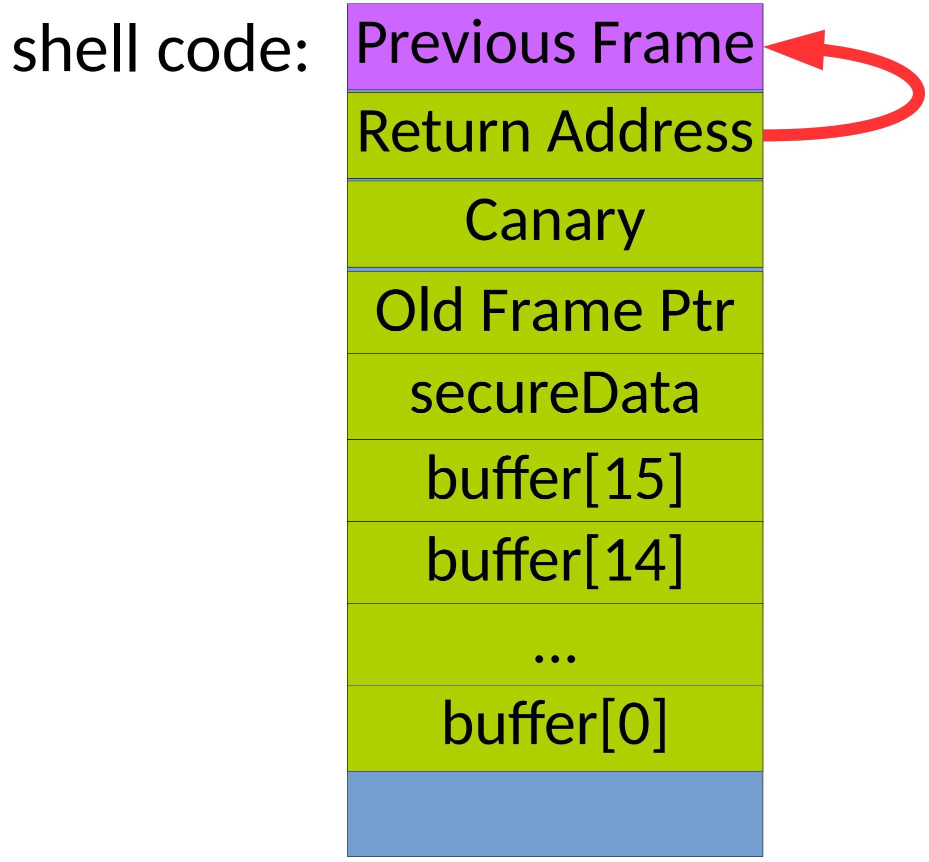


Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries
  - DEP – Data Execution Prevention / W⊕X

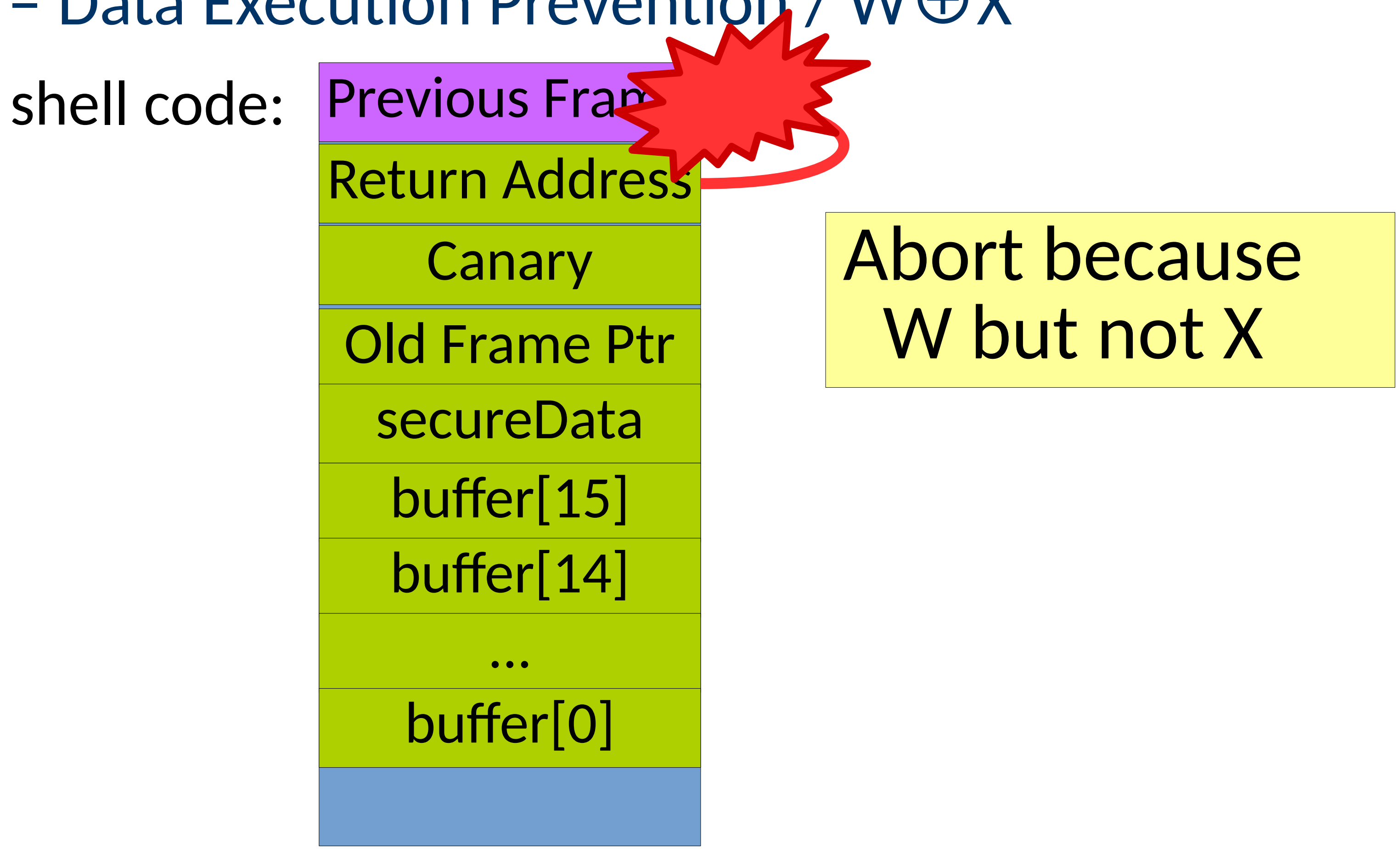
Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries
  - DEP – Data Execution Prevention / W⊕X



Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries
  - DEP – Data Execution Prevention / W⊕X



Control Flow Hijacking

- How can we prevent this basic approach?
  - Stack Canaries
  - DEP – Data Execution Prevention / W⊕X

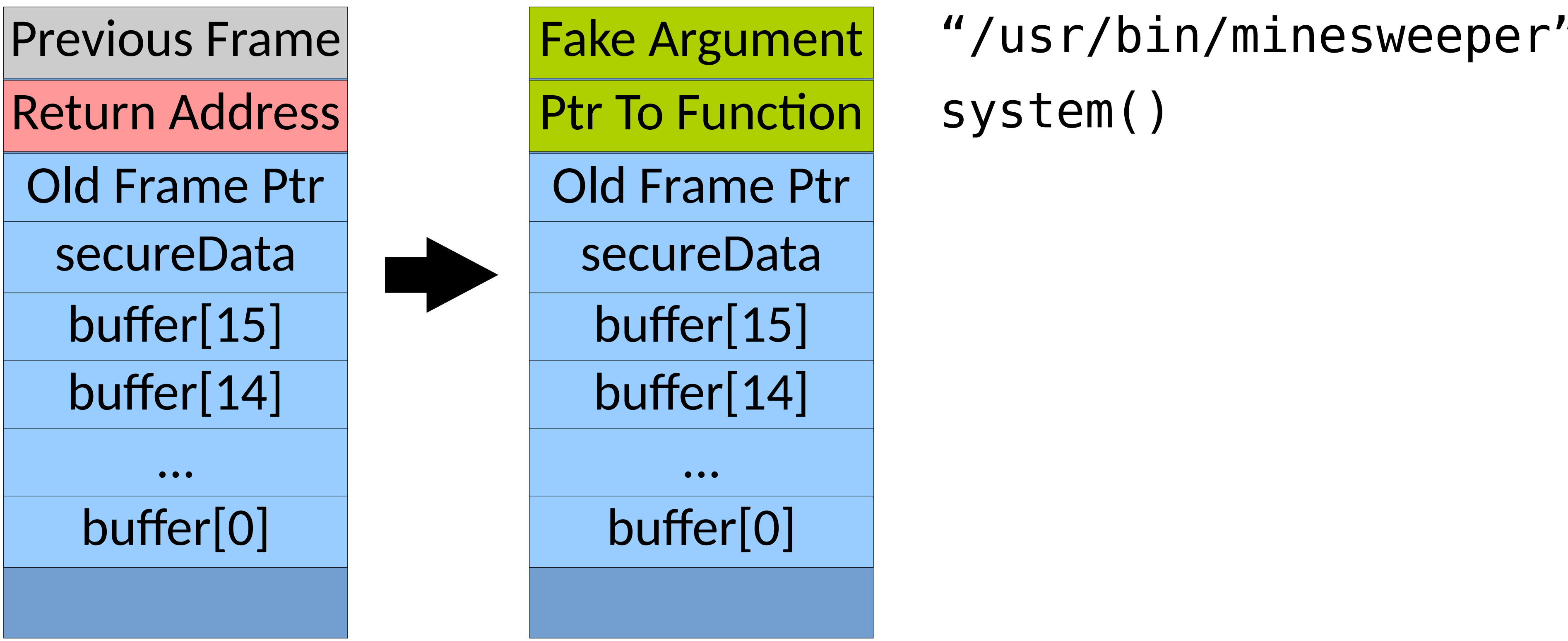
But these are still easily bypassed!

Return to libc Attacks

- Reuse existing code to bypass W⊕X

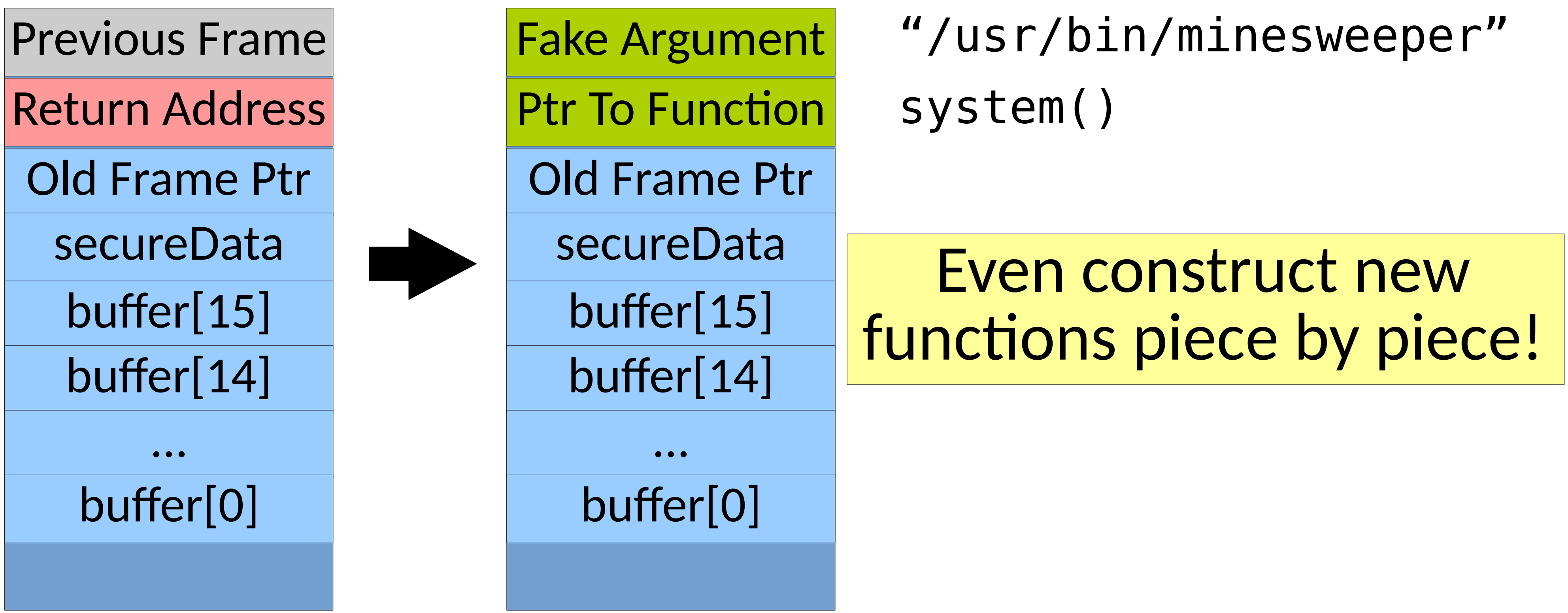
Return to libc Attacks

- Reuse existing code to bypass W⊕X



Return to libc Attacks

- Reuse existing code to bypass W⊕X

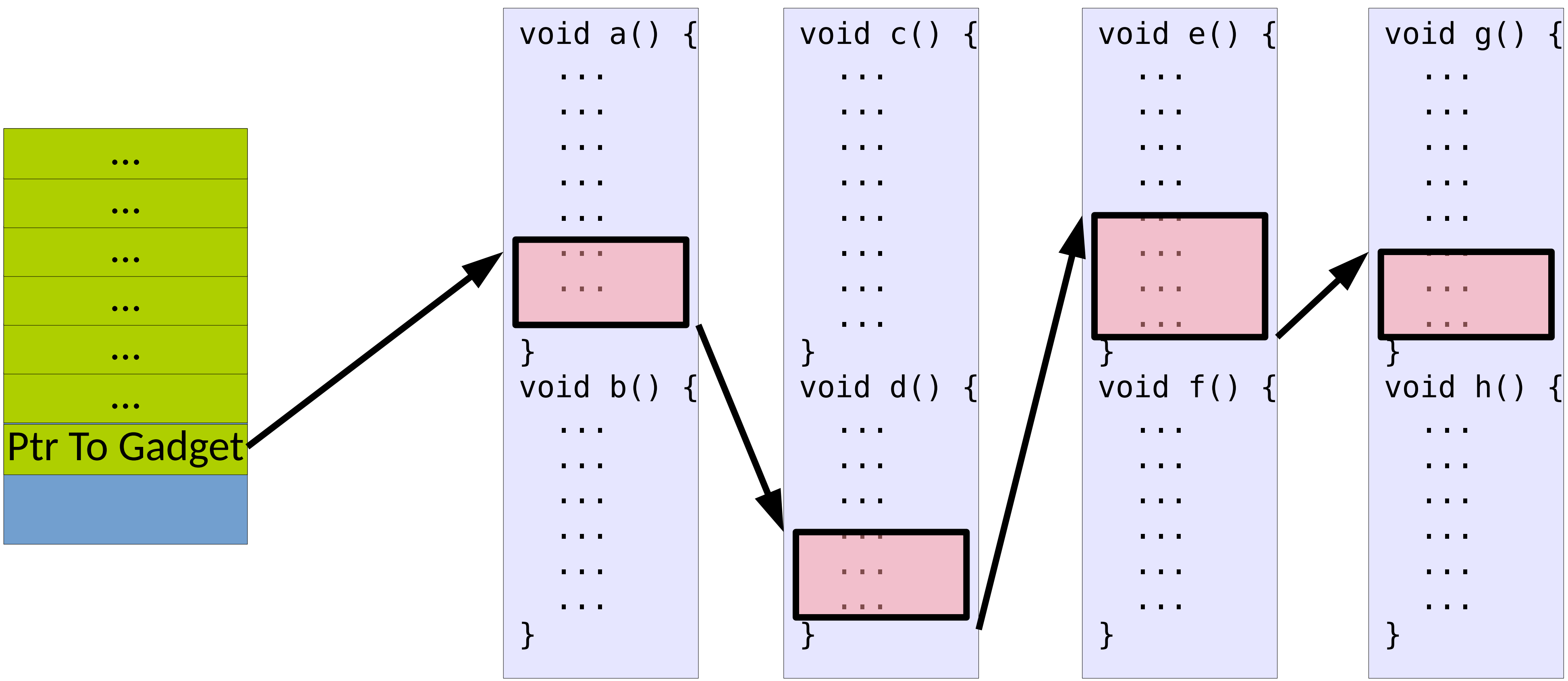


Return to libc Attacks

- Reuse existing code to bypass W⊕X
- Return Oriented Programming
  - Build new functionality from pieces of existing functions

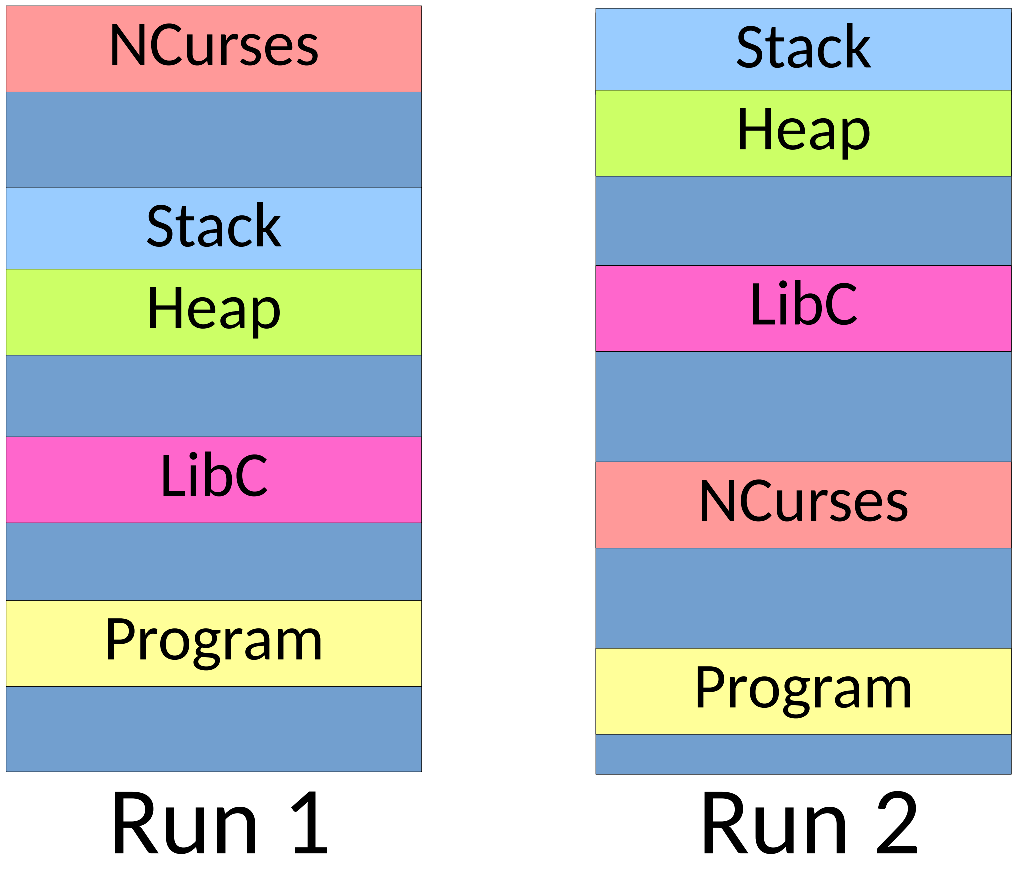
Return to libc Attacks

- Reuse existing code to bypass W⊕X
- Return Oriented Programming
  - Build new functionality from pieces of existing functions



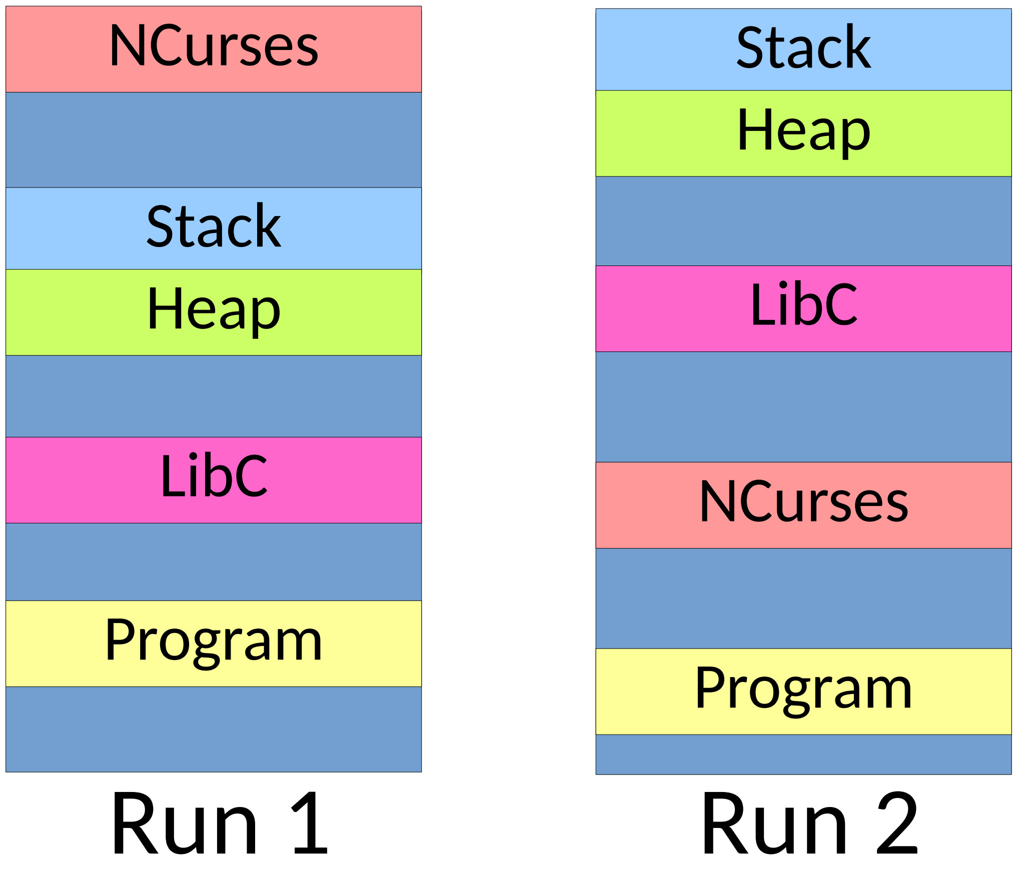
ASLR

- Address Space Layout Randomization
  - You can't use it if you can't find it!



ASLR

- Address Space Layout Randomization
  - You can't use it if you can't find it!



But even this is "easily" broken

Control Flow Integrity

- Restrict indirect control flow to needed targets
  - Jmp */call */ret

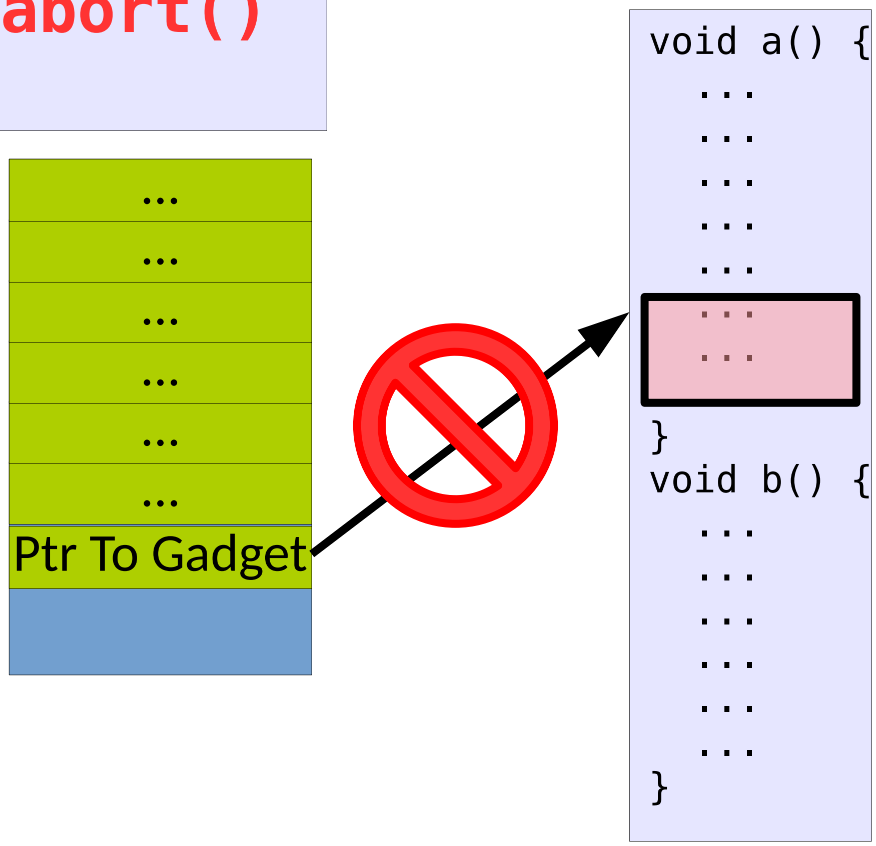
```
foo = ...

foo();
```

Control Flow Integrity

- Restrict indirect control flow to needed targets
  - Jmp */call */ret

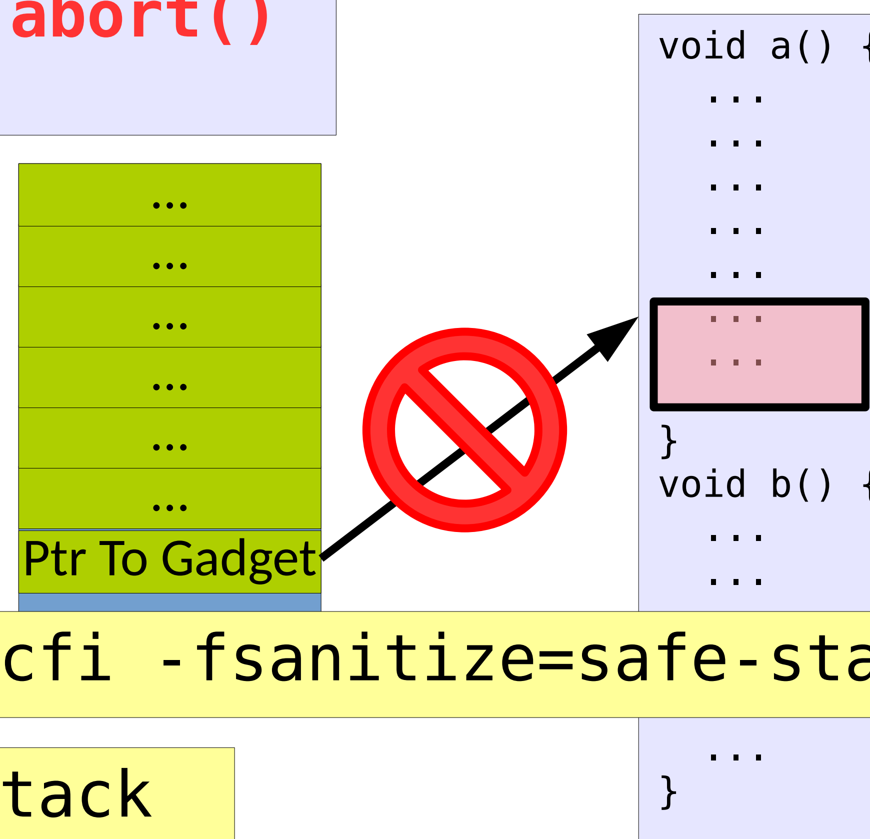
```
foo = ...
if foo not in [...] abort()
foo();
```



Control Flow Integrity

- Restrict indirect control flow to needed targets
  - Jmp */call */ret

```
foo = ...
if foo not in [...] abort()
foo();
```




```
clang -flto -fsanitize=cfi -fsanitize=safe-stack
```

```
clang -fsanitize=safe-stack
```

Memory Safety

- Vulnerabilities come from reading/writing/freeing
  - Out of bounds pointers
  - Dangling pointers

Memory Safety

- Vulnerabilities come from reading/writing/freeing
  - Out of bounds pointers
  - Dangling pointers
- Why doesn't Java face this issue?

Memory Safety

- Vulnerabilities come from reading/writing/freeing
  - Out of bounds pointers
  - Dangling pointers
- Why doesn't Java face this issue?
- Is this intrinsic to languages like C++?
  - Why/Why not?

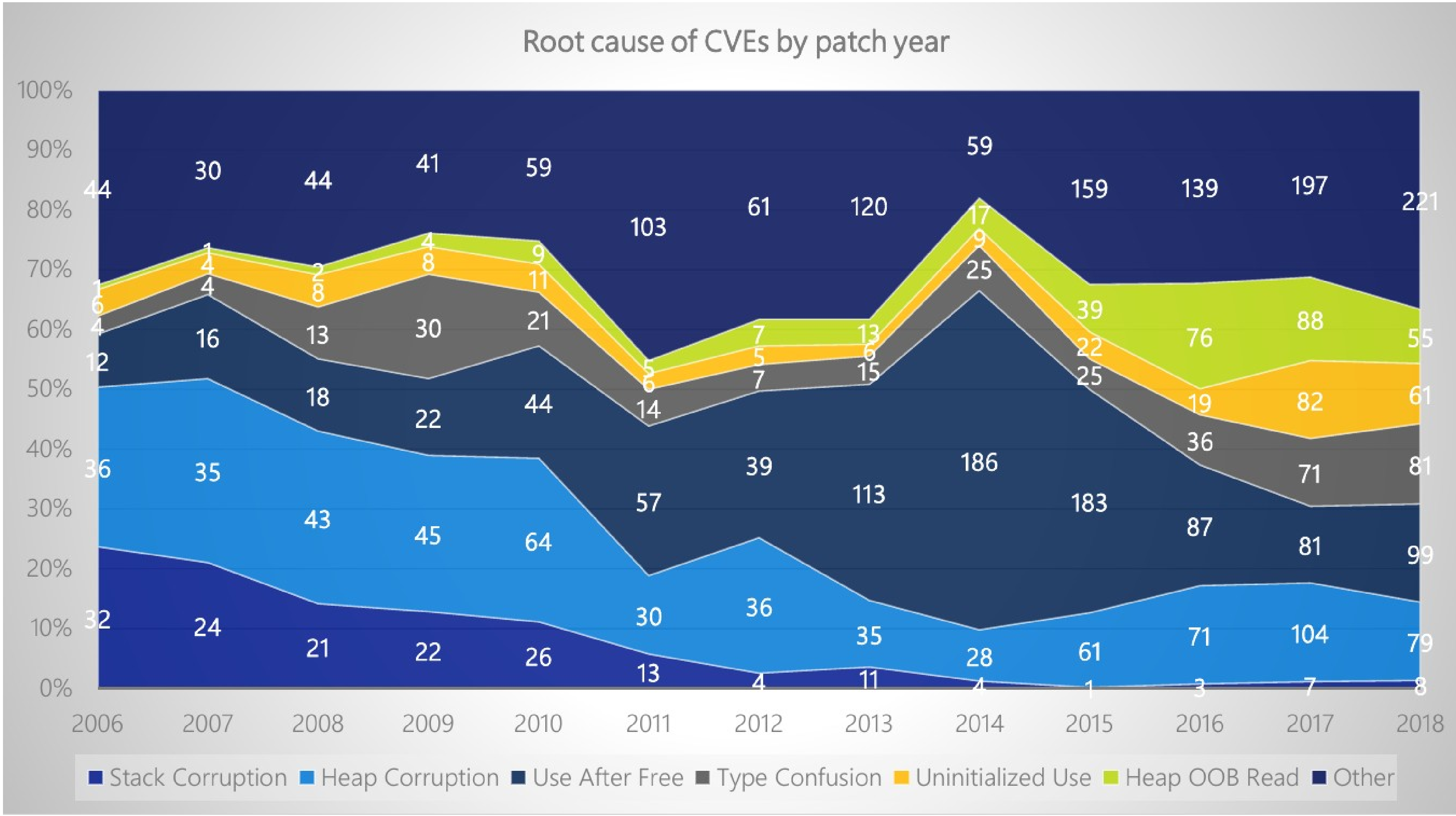
Memory Safety

- Vulnerabilities come from reading/writing/freeing
  - Out of bounds pointers
  - Dangling pointers
- Why doesn't Java face this issue?
- Is this intrinsic to languages like C++?
  - Why/Why not?
- Are these still a real issue?

Memory Safety

- Vulnerabilities come from reading/writing/freeing
  - Out of bounds pointers
  - Dangling pointers
- Why doesn't Java face this issue?
- Is this intrinsic to languages like C++?
  - Why/Why not?
- Are these still a real issue?
  - http://www.symantec.com/security_response/vulnerability.jsp?bid=70332
  - http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0015
  - http://seclists.org/oss-sec/2016/q1/645
  - ...

Root Causes Over Time



[Matt Miller – BlueHat 2019]

Another Case: SQL Injection

SQL – a query language for databases

- Queries like:
  ```
  “SELECT grade,id FROM students
   WHERE name=” + username;
  ```

Another Case: SQL Injection

SQL – a query language for databases

- Queries like:
  “SELECT grade,id FROM students
  WHERE name=” + username;

ID	Name	Grade
0	Alice	92
1	Bob	87
2	Mallory	75

Another Case: SQL Injection

SQL – a query language for databases

- Queries like:
  `“SELECT grade,id FROM students WHERE name=” + username;`

ID	Name	Grade
0	Alice	92
1	Bob	87
2	Mallory	75

- Values for name, grade often come from user input.

Another Case: SQL Injection

SQL – a query language for databases

- Queries like:
```
“SELECT grade,id FROM students
 WHERE name=” + username;
```

ID	Name	Grade
0	Alice	92
1	Bob	87
2	Mallory	75

- Values for name, grade often come from user input.

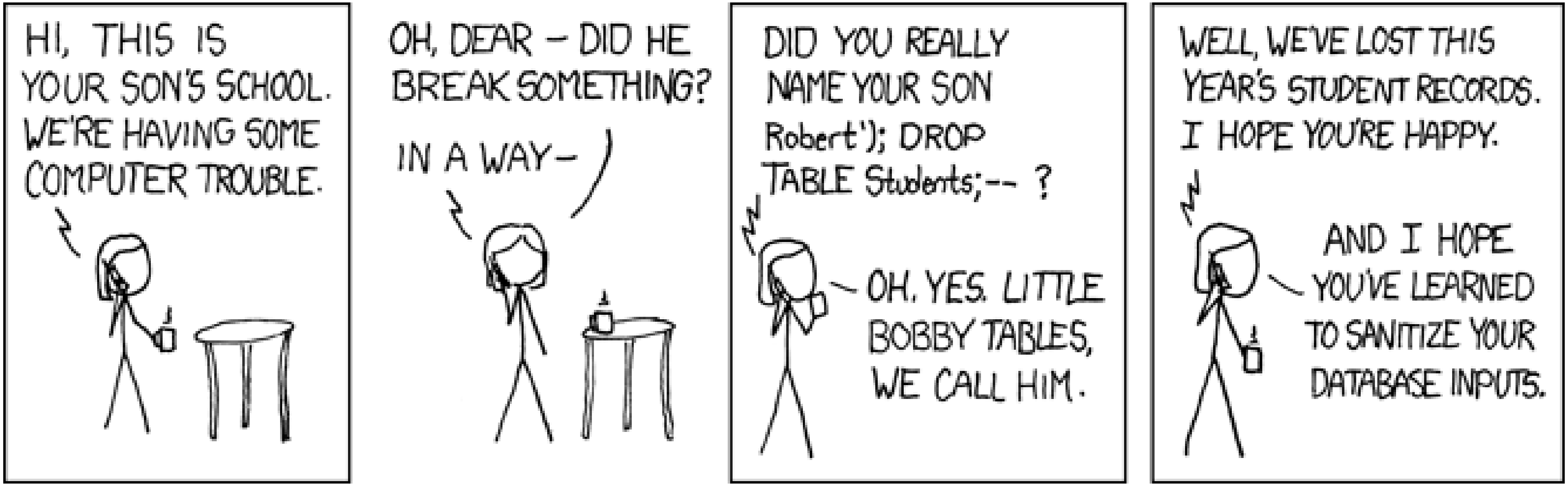
Why is this a problem?

Another Case: SQL Injection

```
username = “'bob'; DROP TABLE students”
```

- What happens?

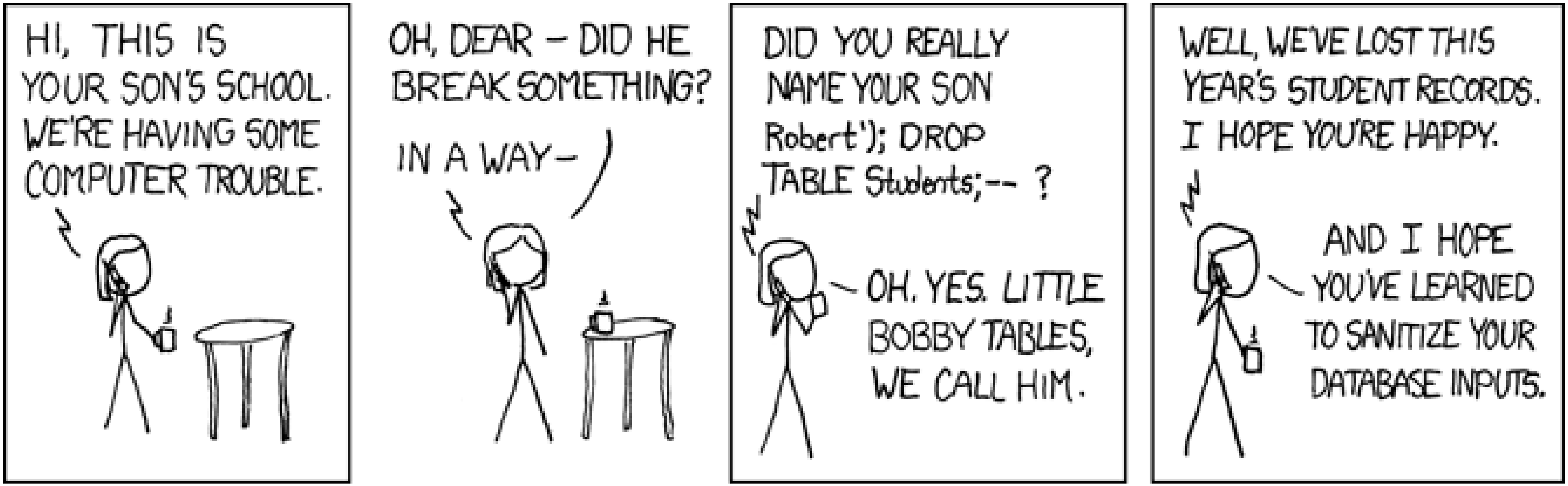
SQL Injection



[http://xkcd.com/327/] [http://bobby-tables.com/]

- The user may include commands in their input!

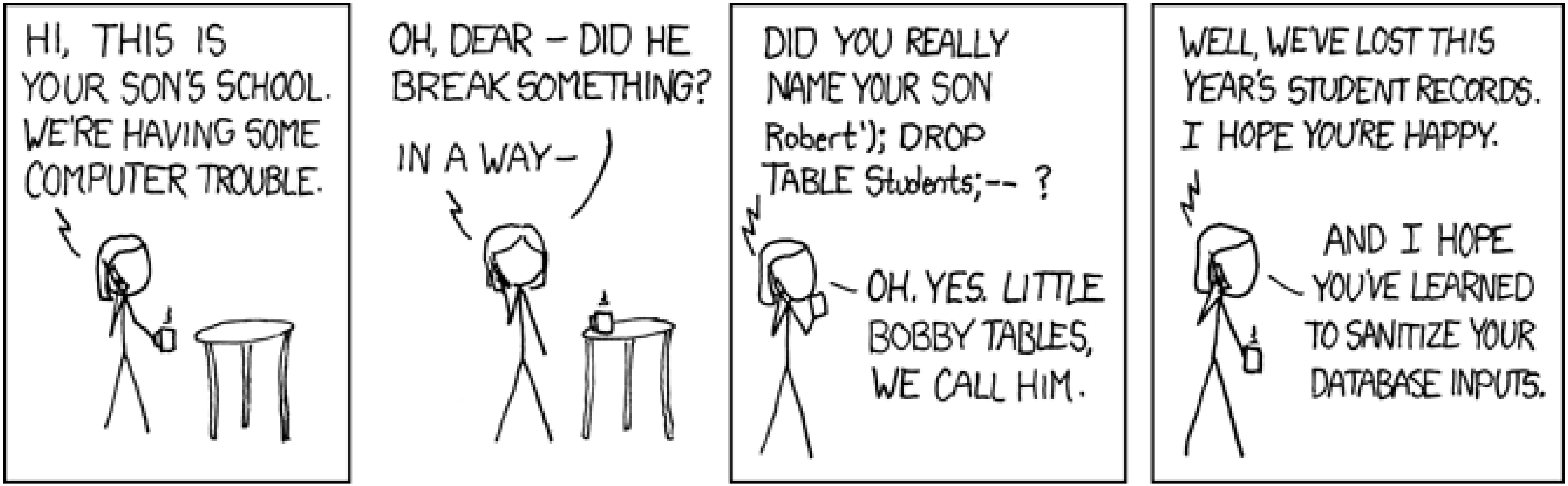
SQL Injection



[http://xkcd.com/327/] [http://bobby-tables.com/]

- The user may include commands in their input!
- Need to *sanitize* the input before use

SQL Injection



[http://xkcd.com/327/] [http://bobby-tables.com/]

- The user may include commands in their input!
- Need to *sanitize* the input before use

How would you prevent this problem?

SQL Injection

- Do not write raw SQL. (examples from bobby-tables.com)

SQL Injection

- Do not write raw SQL. (examples from bobby-tables.com)
  - Sanitizing APIs

SQL Injection

- Do not write raw SQL. (examples from bobby-tables.com)
  - Sanitizing APIs

```
List<Person>; people = //user input
Connection connection = DriverManager.getConnection(...);
connection.setAutoCommit(false);
try {
    PreparedStatement statement = connection.prepareStatement(
            "UPDATE people SET lastName = ?, age = ? WHERE id = ?");
    for (Person person : people){
        statement.setString(1, person.getLastName());
        statement.setInt(2, person.getAge());
        statement.setInt(3, person.getId());
        statement.execute();
    }
    connection.commit();
} catch (SQLException e) {
    connection.rollback();
}
```

SQL Injection

- Do not write raw SQL. (examples from bobby-tables.com)
  - Sanitizing APIs

```
EntityManager em = getEntityManager();
Query query = em.createNativeQuery("SELECT E.* from EMP E, ADDRESS A
                                    WHERE E.EMP_ID = A.EMP_ID AND A.CITY = ?",
                                    Employee.class);
query.setParameter(1, "Ottawa");
List<Employee> employees = query.getResultList();
```

SQL Injection

- Do not write raw SQL. (examples from bobby-tables.com)
  - Sanitizing APIs
  - ORMs (to *some* degree!) [Fixing SQL Injection w/ Hibernate]

SQL Injection

- Do not write raw SQL. (examples from bobby-tables.com)
  - Sanitizing APIs
  - ORMs (to *some* degree!) [Fixing SQL Injection w/ Hibernate]

```
String name = //user input
int age = //user input
Session session = //...
Query query = session.createQuery(
        "from People where lastName = :name and age > :age");
query.setString("name", name);
query.setInteger("age", age);
Iterator people = query.iterate();
```

SQL Injection

- Do not write raw SQL. (examples from bobby-tables.com)
  - Sanitizing APIs
  - ORMs (to *some* degree!) [Fixing SQL Injection w/ Hibernate]

```
String name = //user input
int age = //user input
Session session = //…
Query query = session.createQuery(
        "from People where lastName = :name and age > :age");
query.setString("name", name);
query.setInteger("age", age);
Iterator people = query.iterate();
```

- Use abstractions that design error away if possible!
  - Applies whenever you generate code in another language (think web apps)

Side Channels

- So far we have looked for ways to *directly* violate CIA

Side Channels

- So far we have looked for ways to *directly* violate CIA
  - Execute code
  - Explicitly broadcast a value
  - …

Side Channels

- So far we have looked for ways to *directly* violate CIA
  - Execute code
  - Explicitly broadcast a value
  - ...
- An attacker can *indirectly* violate CIA by inferring sensitive information

Side Channels

- So far we have looked for ways to *directly* violate CIA
  - Execute code
  - Explicitly broadcast a value
  - …
- An attacker can *indirectly* violate CIA by inferring sensitive information
  - *Side channel attacks* can infer secret information about a system based on implementation details

Side Channels

- So far we have looked for ways to *directly* violate CIA
  - Execute code
  - Explicitly broadcast a value
  - ...
- An attacker can *indirectly* violate CIA by inferring sensitive information
  - *Side channel attacks* can infer secret information about a system based on implementation details
  - These leaks can be present even for algorithms that are mathematically correct

Side Channels

- So far we have looked for ways to *directly* violate CIA
  - Execute code
  - Explicitly broadcast a value
  - …
- An attacker can *indirectly* violate CIA by inferring sensitive information
  - *Side channel attacks* can infer secret information about a system based on implementation details
  - These leaks can be present even for algorithms that are mathematically correct
  - Leaks can come from:
    Output, Timing (compute, cache, MDS,…), Power, Sound, Light, …

Side Channels

- Consider code that directly leaks a sensitive boolean

```
def very_stupid(greeting, sensitive):
    …
    log_to_nonsensitive(sensitive)
    …
```

Side Channels

- Consider code that directly leaks a sensitive boolean

```
def very_stupid(greeting, sensitive):
    ...
    log_to_nonsensitive(sensitive)
    ...
```

  - This could be tweaked to become an indirect leak

```
def still_bad(greeting, sensitive):
    ...
    if sensitive:
        log_to_nonsensitive(greeting)
    ...
```

Side Channels

- Consider code that directly leaks a sensitive boolean

```
def very_stupid(greeting, sensitive):
    ...
    log_to_nonsensitive(sensitive)
    ...
```

  - This could be tweaked to become an indirect leak

```
def still_bad(greeting, sensitive):
    ...
    if sensitive:
        log_to_nonsensitive(greeting)
    ...
```

  - The ***value*** of the sensitive information can be inferred by the ***existence*** of the nonsensitive information!

Side Channels

- Any difference in behavior between sensitive and nonsensitive tasks can be measured and used

Side Channels

- Any difference in behavior between sensitive and nonsensitive tasks can be measured and used

```
def subtly_bad(greeting, sensitive):
    ...
    if sensitive:
        expensive_computation()
    log_to_nonsensitive(greeting)
    ...
```

Side Channels

- Any difference in behavior between sensitive and nonsensitive tasks can be measured and used

```
def subtly_bad(greeting, sensitive):
    ...
    if sensitive:
        expensive_computation()
    log_to_nonsensitive(greeting)
    ...
```

This has been the downfall of crypto implementations!

Side Channels

- Any difference in behavior between sensitive and nonsensitive tasks can be measured and used

```
def subtly_bad(greeting, sensitive):
    …
    if sensitive:
        expensive_computation()
    log_to_nonsensitive(greeting)
    …
```

```
def deviously_bad(greeting, sensitive):
    …
    if sensitive:
        a[not_in_cache] = …
    log_to_nonsensitive(greeting)
    …
```

Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```
if x < array1.size:
    y = array2[array1[x] * 4096]
```

Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```
if x < array1.size:
    y = array2[array1[x] * 4096]
```

Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```
if x < array1.size:
    y = array2[array1[x] * 4096]
```

When the condition is *true*, array1[x] will be in bounds

Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```
if x < array1.size:
      y = array2[array1[x] * 4096]
```

When the condition is *true*, array1[x] will be in bounds

When the condition is *false*, array1[x] can be anywhere

Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```
if x < array1.size:
      y = array2[array1[x] * 4096]
```

When the condition is *true*, array1[x] will be in bounds

When the condition is *false*, array1[x] can be anywhere

An attacker can

Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```
if x < array1.size:
    y = array2[array1[x] * 4096]
```

When the condition is *true*, array1[x] will be in bounds

When the condition is *false*, array1[x] can be anywhere

An attacker can
1) make array1[x] point to sensitive data

Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```
if x < array1.size:
      y = array2[array1[x] * 4096]
```

When the condition is *true*, array1[x] will be in bounds

When the condition is *false*, array1[x] can be anywhere

An attacker can
1) make array1[x] point to sensitive data
2) train the branch to speculate true

Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differ

The sensitive data is speculatively read and used!

```
if x < array1.size:
    y = array2[array1[x] * 4096]
```

When the condition is *true*, array1[x] will be in bounds

When the condition is *false*, array1[x] can be anywhere

An attacker can
1) make array1[x] point to sensitive data
2) train the branch to speculate true

Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```
if x < array1.size:
    y = array2[array1[x] * 4096]
```

When the condition is *true*, array1[x] will be in bounds

When the condition is *false*, array1[x] can be anywhere

An attacker can
1) make array1[x] point to sensitive data
2) train the branch to speculate true
3) extract the data through a 1-hot encoding in the time to access elements of array2 (or a buffer sharing the cache mapping of array2)

Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```
if x < array1.size:
    y = array2[array1[x] * 4096]
```

```
# foo is a function pointer
foo()
```

Foo can be trained to speculate to an arbitrary gadget!

Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```
if x < array1.size:
    y = array2[array1[x] * 4096]
```

```
# foo is a function pointer
foo()
```

```
def foo():
    return
```

Return targets can be trained to speculate to gadgets!

Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```
if x < array1.size:
    y = array2[array1[x] * 4096]
```

```
# foo is a function pointer
foo()
```

```
def foo():
    return
```

Note: This means that ROP gadgets can once again be used!
Newer compiler options can mitigate but not remove the challenge

[Speculative Load Hardening in LLVM] clang -mretpoline -mspeculative-load-hardening ...

Side Channels

- This is the fundamental premise behind Spectre and generic MDS based attacks
  - Spectre worked by mistraining speculation & then measuring timing differences

```
if x < array1.size:
    y = array2[array1[x] * 4096]
```

```
# foo is a function pointer
foo()
```

```
def foo():
    return
```

  - MDS attacks leverage other CPU artifacts to achieve similar goals (line buffers, ports, etc.)
    - Contention on any resource affects timing

A Subtle Problem in General

- The problems may be much more subtle:

  User A can read files X,Y,Z and write to S,T
  User B can read files X,Y,S and write to Z,T

A Subtle Problem in General

- The problems may be much more subtle:

User A can read files X,Y,Z and write to S,T
User B can read files X,Y,S and write to Z,T

How can we ensure that no information
from A is ever written to Z?

A Subtle Problem in General

- The problems may be much more subtle:

User A can read files X,Y,Z and write to S,T
User B can read files X,Y,S and write to Z,T

How can we ensure that no information
from A is ever written to Z?

Can you envision a scenario
that creates this problem?

A Subtle Problem in General

- The problems may be much more subtle:

    User A can read files X,Y,Z and write to S,T
    User B can read files X,Y,S and write to Z,T

    How can we ensure that no information
    from A is ever written to Z?

- Care may be required to enforce *access control policies*

A Subtle Problem in General

- The problems may be much more subtle:

  User A can read files X,Y,Z and write to S,T
  User B can read files X,Y,S and write to Z,T

  How can we ensure that no information
  from A is ever written to Z?

- Care may be required to enforce *access control policies*
  - *Discretionary* access control – owner determines access

A Subtle Problem in General

- The problems may be much more subtle:

  User A can read files X,Y,Z and write to S,T
  User B can read files X,Y,S and write to Z,T

  How can we ensure that no information from A is ever written to Z?

- Care may be required to enforce *access control policies*
  - Discretionary access control – owner determines access
  - *Mandatory* access control – clearance determines access

Assuring Security

- Make risky operations someone else's job
  - e.g. Google Checkout, PayPal, Amazon, etc.

Assuring Security

- Make risky operations someone else's job
  - e.g. Google Checkout, PayPal, Amazon, etc.
- Define rigorous security policies
  - What are your CIA security criteria?

Assuring Security

- Make risky operations someone else's job
  - e.g. Google Checkout, PayPal, Amazon, etc.
- Define rigorous security policies
  - What are your CIA security criteria?
- Follow secure design & coding policies
  - And include them in your review criteria

Assuring Security

- Make risky operations someone else's job
  - e.g. Google Checkout, PayPal, Amazon, etc.
- Define rigorous security policies
  - What are your CIA security criteria?
- Follow secure design & coding policies
  - And include them in your review criteria
  - Apple secure coding policies
  - CERT Top 10 Practices
  - Mitre Mitigation Strategies

Assuring Security

- Make risky operations someone else's job
  - e.g. Google Checkout, PayPal, Amazon, etc.
- Define rigorous security policies
  - What are your CIA security criteria?
- Follow secure design & coding policies
  - And include them in your review criteria
- Formal certification

Assuring Security

- Make risky operations someone else's job
  - e.g. Google Checkout, PayPal, Amazon, etc.
- Define rigorous security policies
  - What are your CIA security criteria?
- Follow secure design & coding policies
  - And include them in your review criteria
- Formal certification
- Follow established security workflows (OWASP, BSIMM, ...)

Common Proactive Approaches

How are these techniques applied?

Common Proactive Approaches

How are these techniques applied?

- Security must be part of design
  - Prepared Statements, Safe Arrays, etc.

Common Proactive Approaches

How are these techniques applied?

- Security must be part of design
  - Prepared Statements, Safe Arrays, etc.
- Regular security audits
  - Retrospective analysis & suggestions

Common Proactive Approaches

How are these techniques applied?

- Security must be part of design
  - Prepared Statements, Safe Arrays, etc.
- Regular security audits
  - Retrospective analysis & suggestions
- Penetration testing (Pen Testing)
  - Can someone skilled break it?

When you find a vulnerability

- Reporting security vulnerabilities is good

When you find a vulnerability

- Reporting security vulnerabilities is good
- Making them public immediately is not

When you find a vulnerability

- Reporting security vulnerabilities is good
- Making them public immediately is not
- ***Responsible disclosure*** policies govern the trade off between allowing a fix to be deployed & awareness

When you find a vulnerability

- Reporting security vulnerabilities is good
- Making them public immediately is not
- Responsible disclosure policies govern the trade off between allowing a fix to be deployed & awareness
  - e.g. Google standard 90 day window
    7 month window for Spectre due to severity
    …

Security Overall

- Security is now a pressing concern for all software

Security Overall

- Security is now a pressing concern for all software
  - Old software was designed in an era of naiveté and is often vulnerable/broken

Security Overall

- Security is now a pressing concern for all software
  - Old software was designed in an era of naiveté and is often vulnerable/broken
  - New software is built to perform sensitive operations in a multiuser and networked environment.

Security Overall

- Security is now a pressing concern for all software
  - Old software was designed in an era of naiveté and is often vulnerable/broken
  - New software is built to perform sensitive operations in a multiuser and networked environment.

Not planning for security concerns from the beginning is a broken approach to development