CMPT 886
Program Analysis & Reliabilty

# Software Security & Memory Safety

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

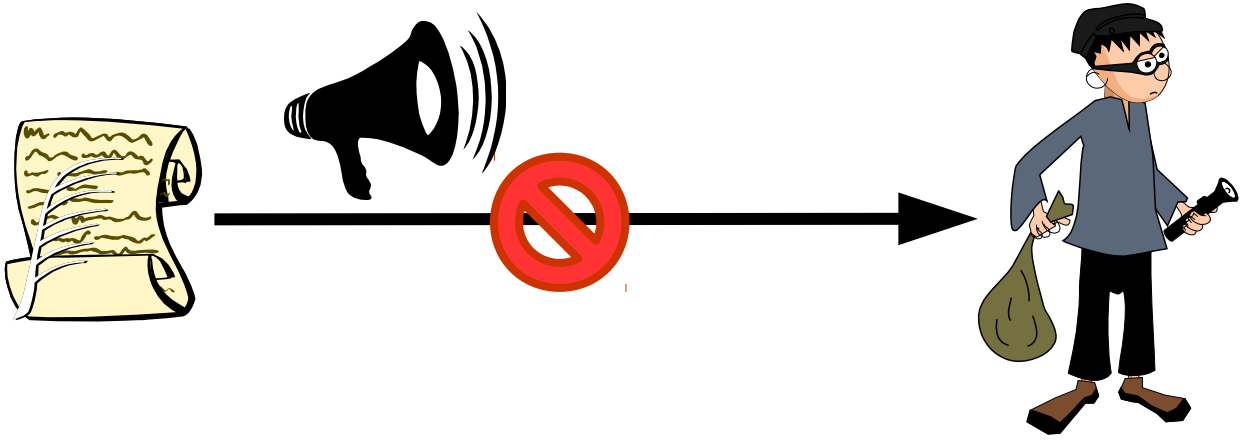
So what are the desired properties?

# Security in General

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

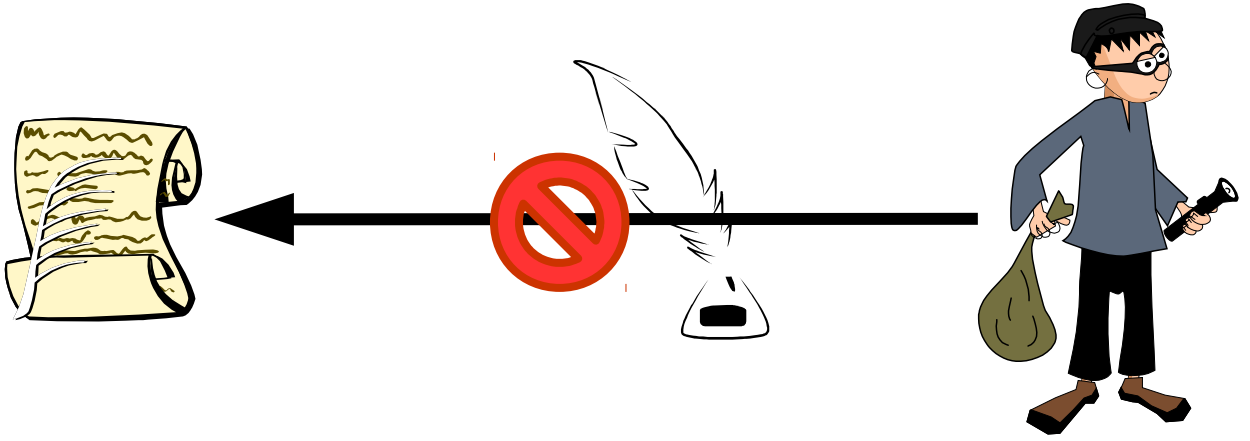
# Security in General

- ***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



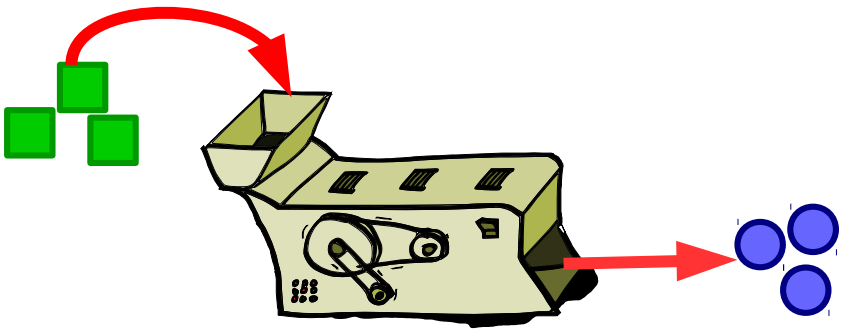
# Security in General

- ***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



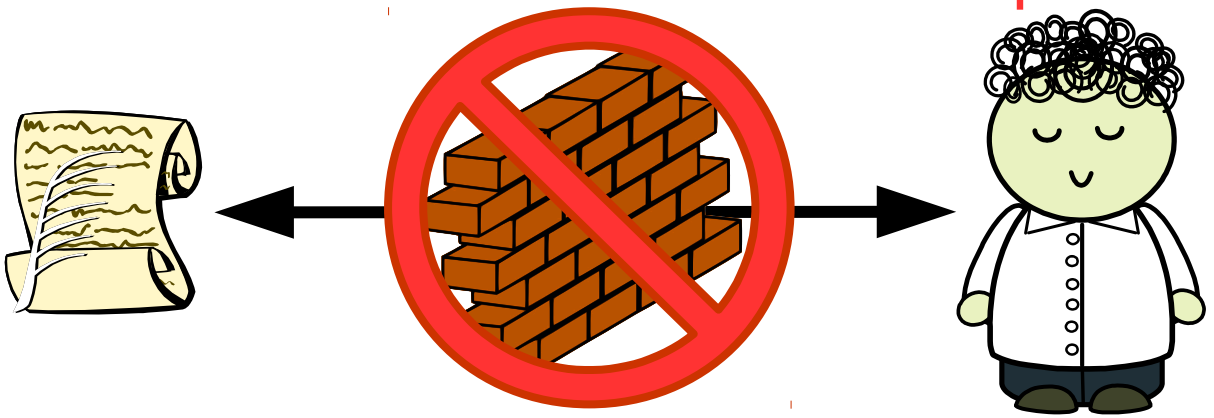
# Security in General

- ***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
    - Do what is expected



# Security in General

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



# Security in Software

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

# Security in Software

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

# Security in Software

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

# Security in Software

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

# Security in Software

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

# Security in Software

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

All create attack vectors
for a malicious adversary

# Software Security

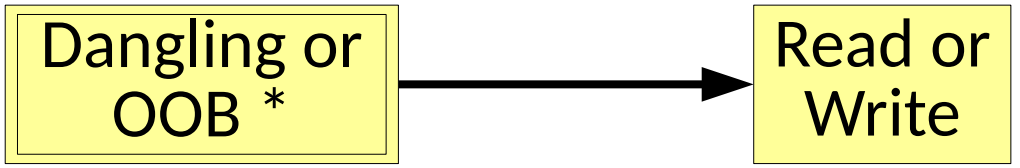
- *Unsafe memory* accesses are a longstanding vector
  - Memory Safety [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]

# Software Security

- *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]

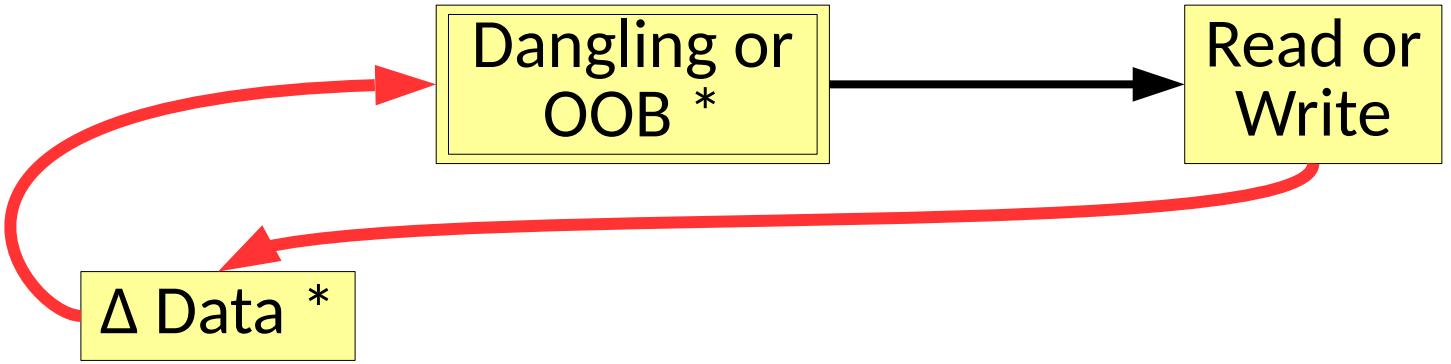
# Software Security

- *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]



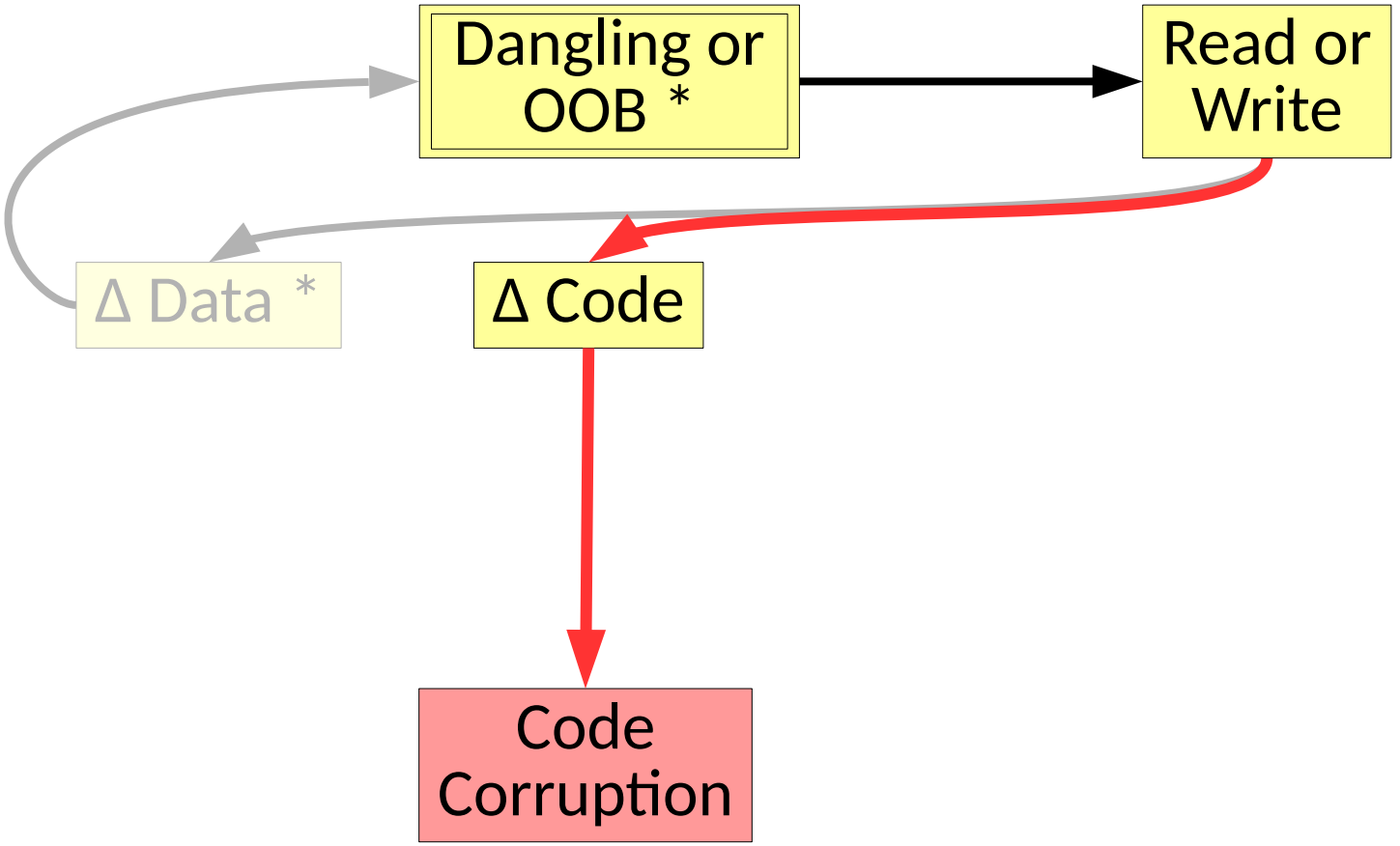
# Software Security

- *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]



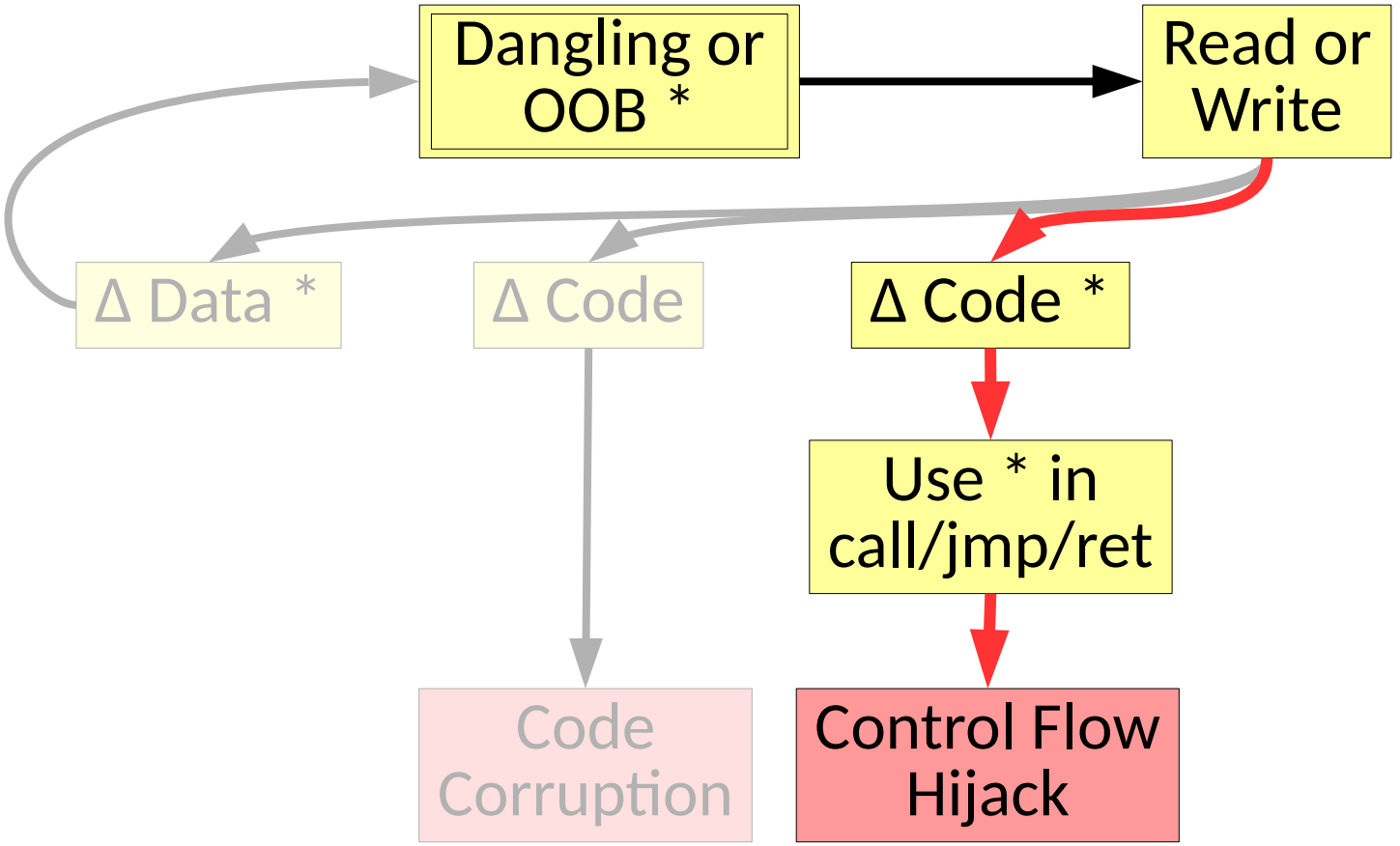
# Software Security

- *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]



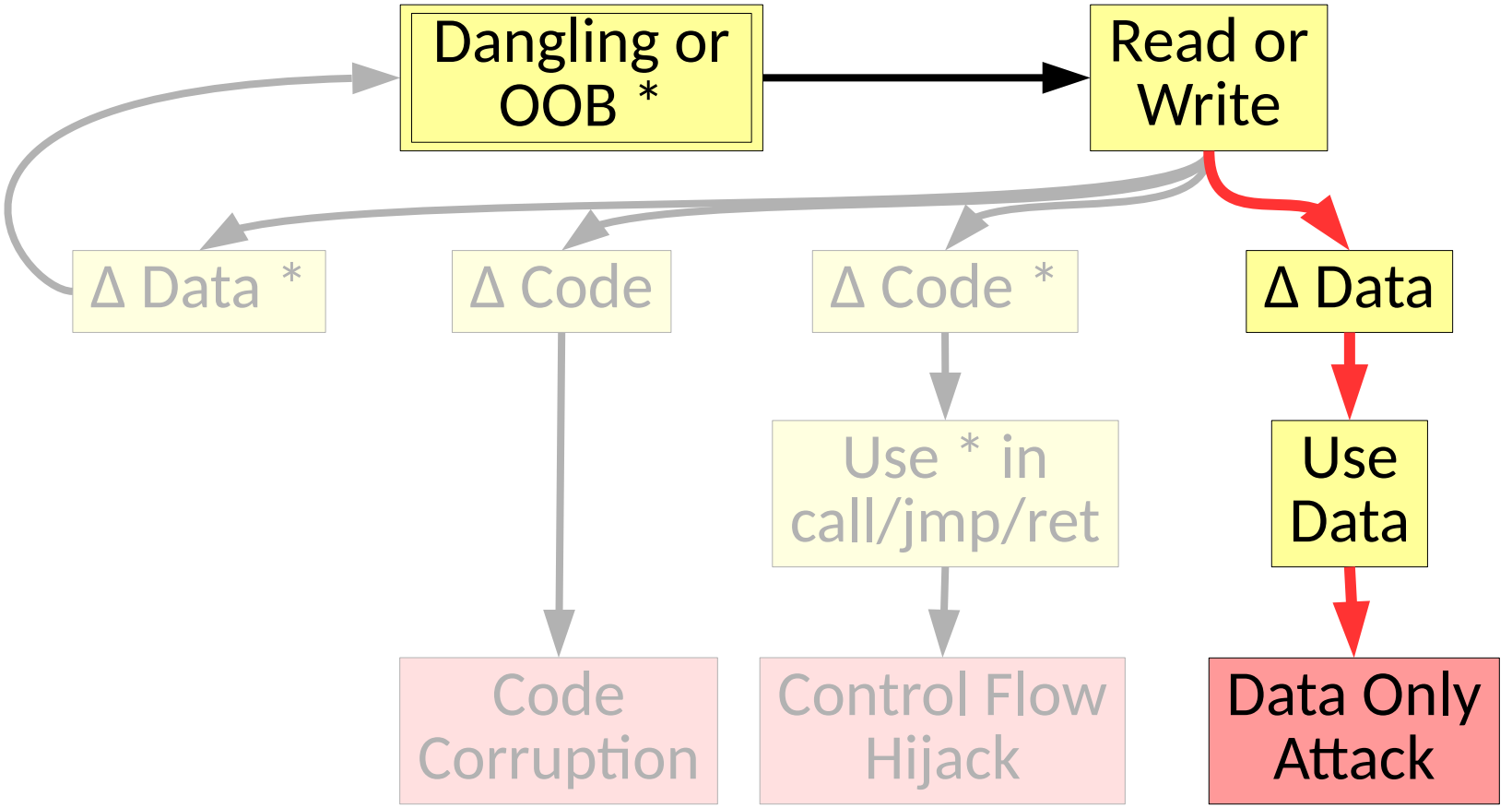
# Software Security

- *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]



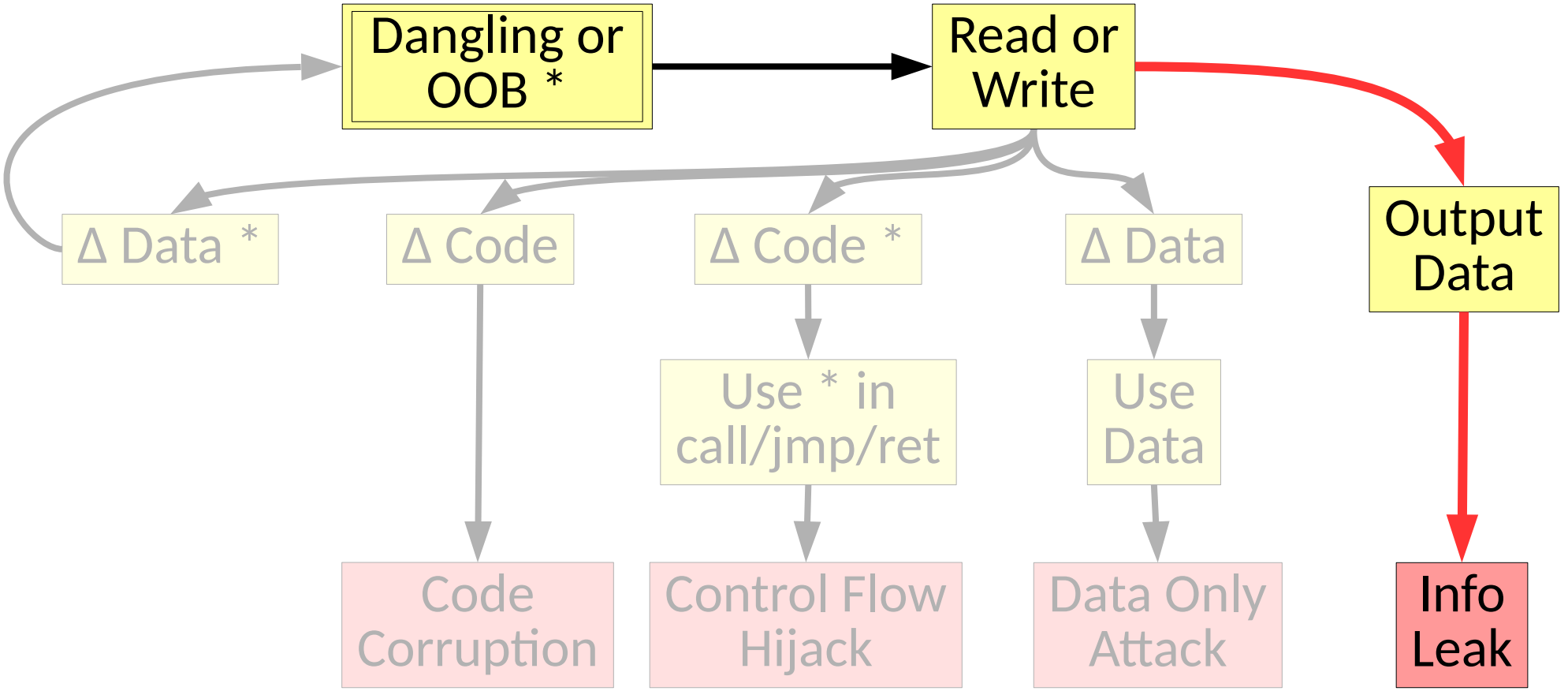
# Software Security

- *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]

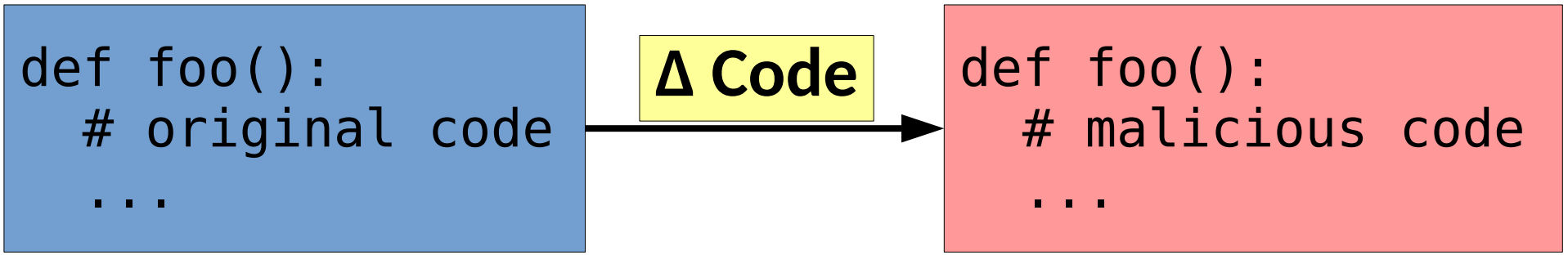


# Software Security

- *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]

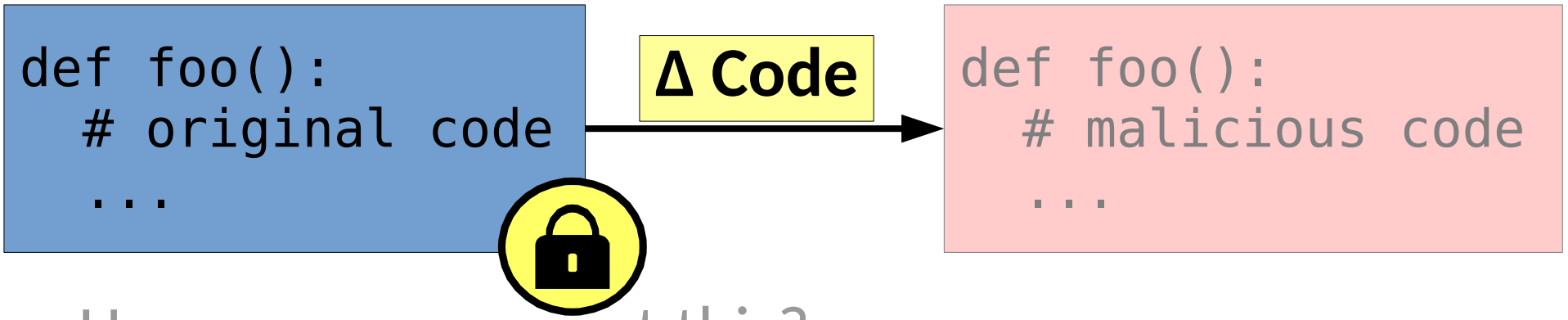


# Code Corruption



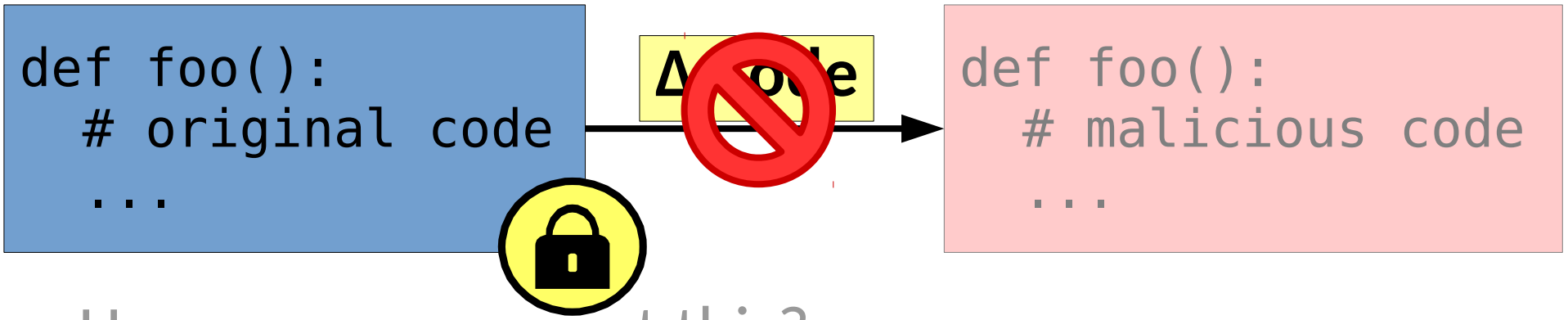

- How can we prevent this?

# Code Corruption



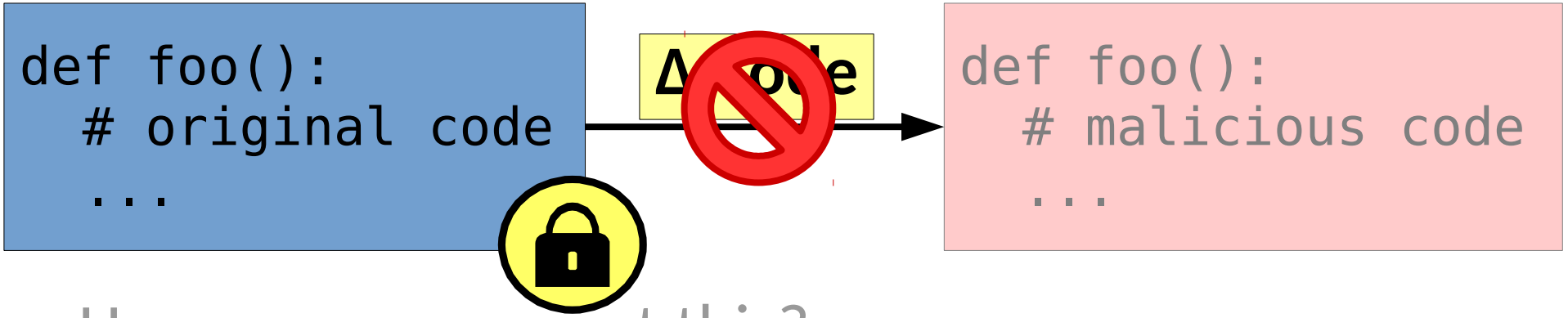

- How can we prevent this?

# Code Corruption




- How can we prevent this?

# Code Corruption



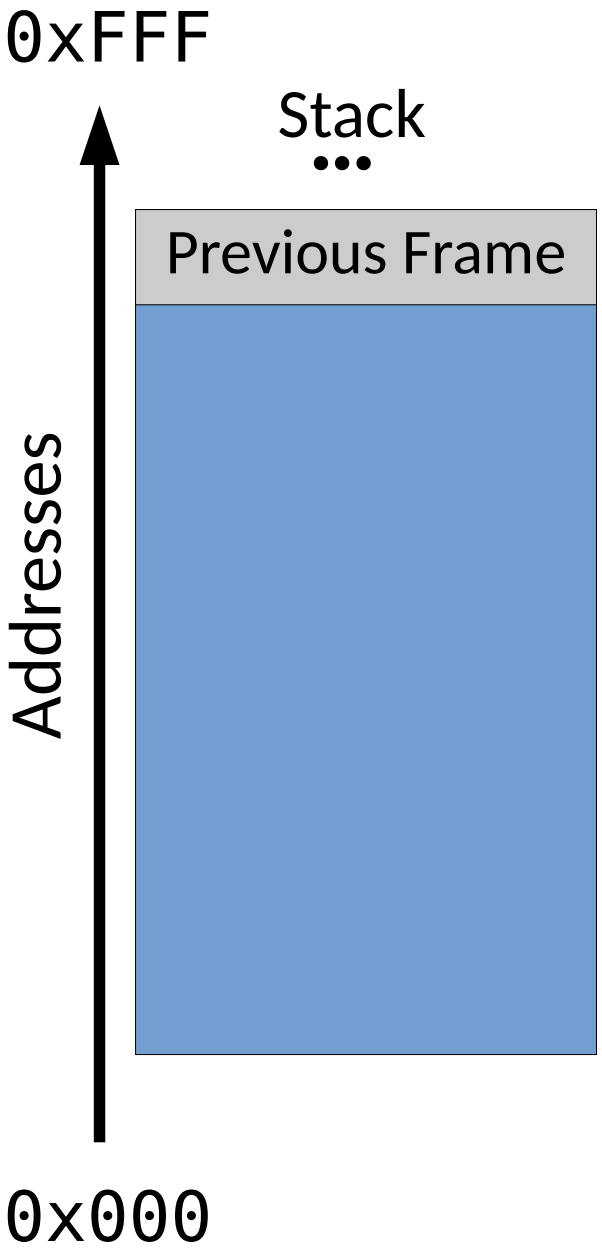

- How can we prevent this?
- What problems could this solution create?

(Might you want executable data?)

# Control Flow Hijacking

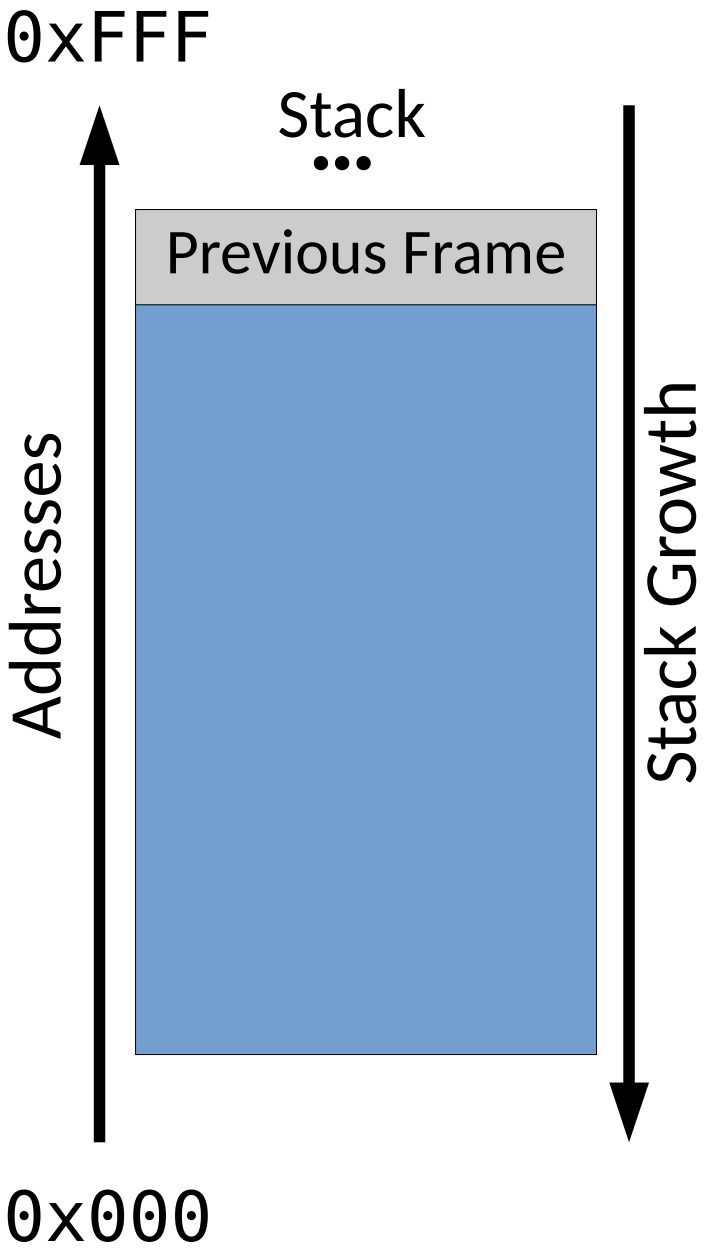
```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

# Control Flow Hijacking



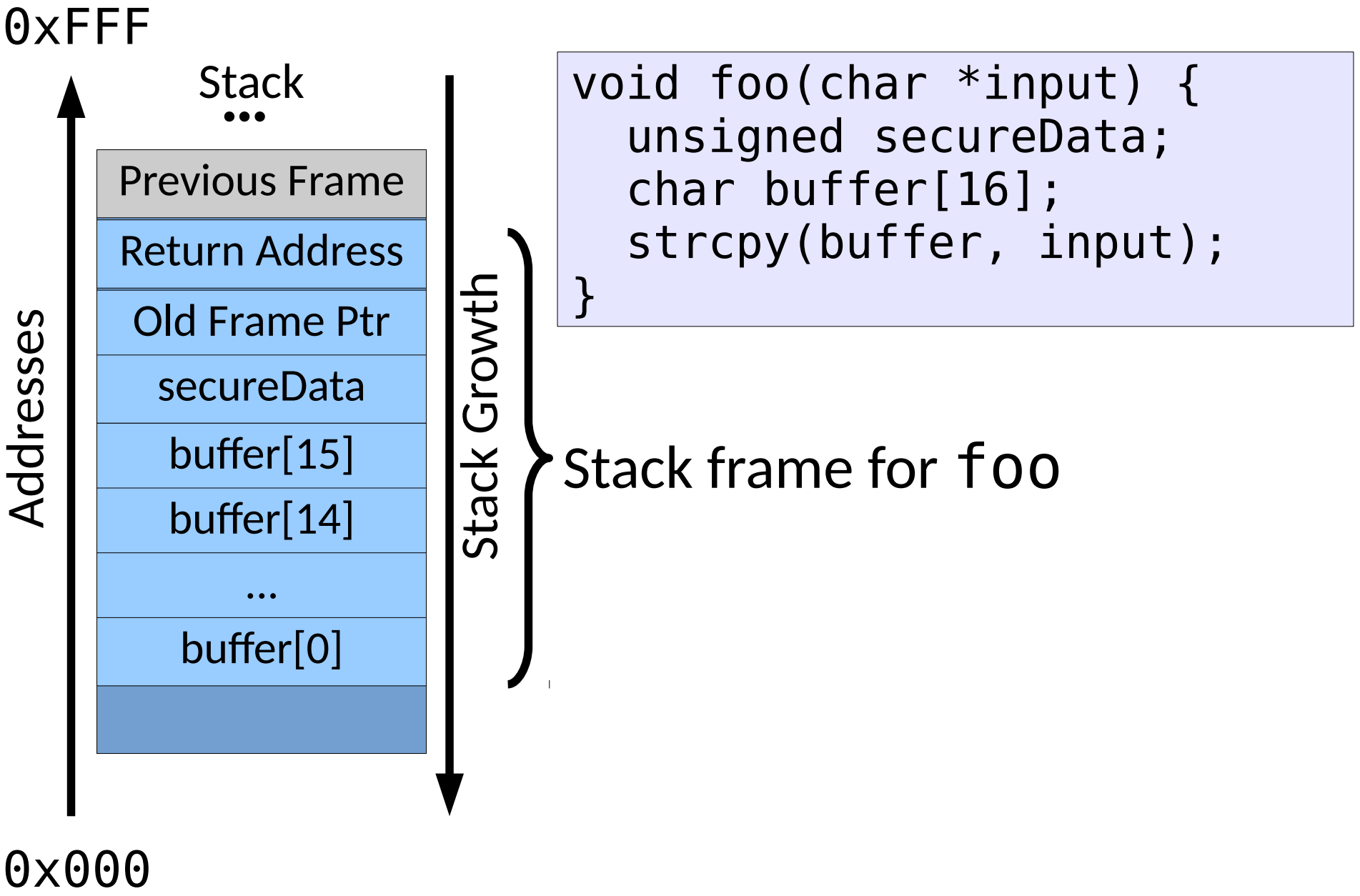

```
void foo(char *input) {
  unsigned secureData;
  char buffer[16];
  strcpy(buffer, input);
}
```

# Control Flow Hijacking




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

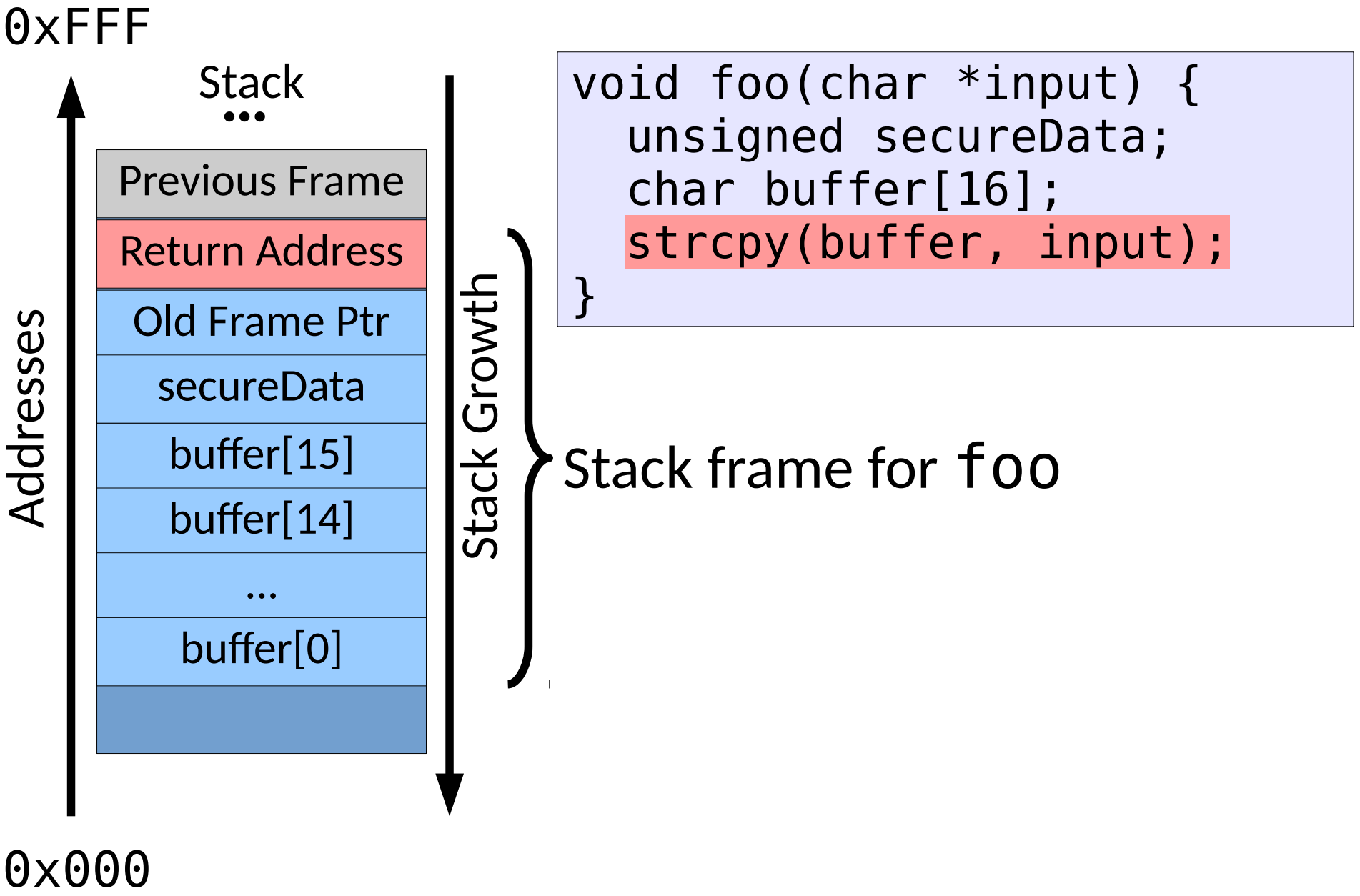
# Control Flow Hijacking




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

Stack frame for `foo`

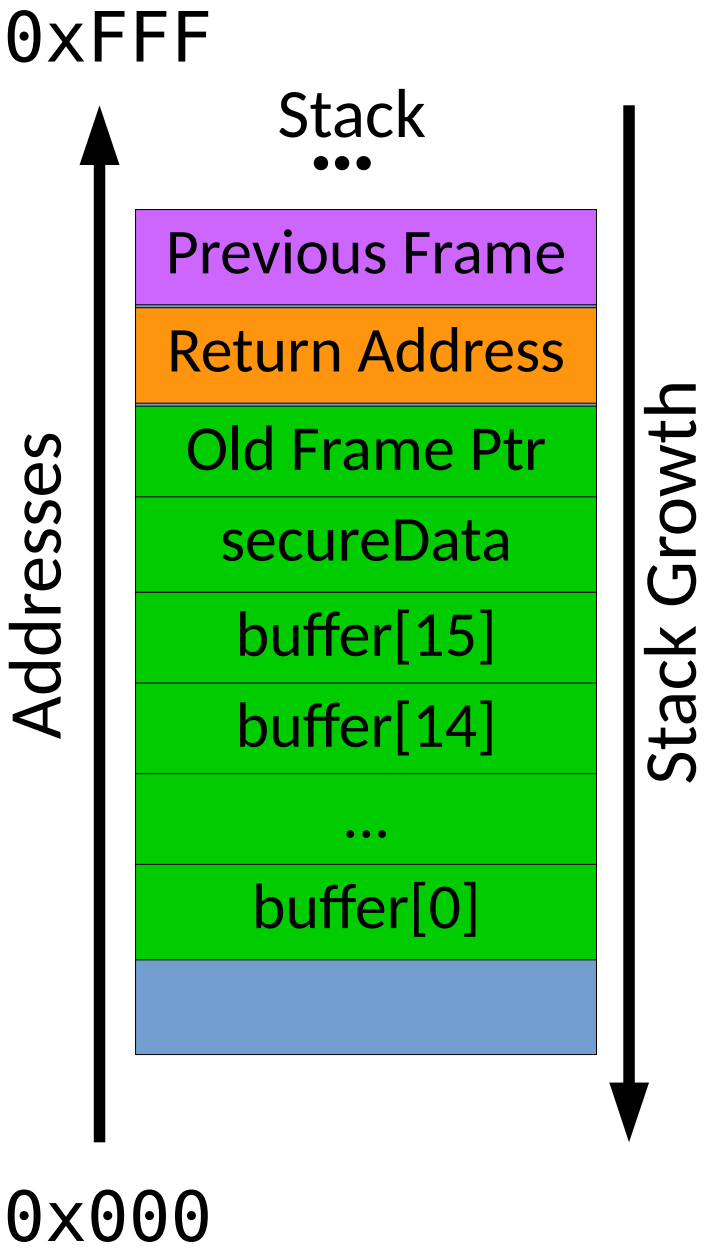
# Control Flow Hijacking




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

Stack frame for `foo`

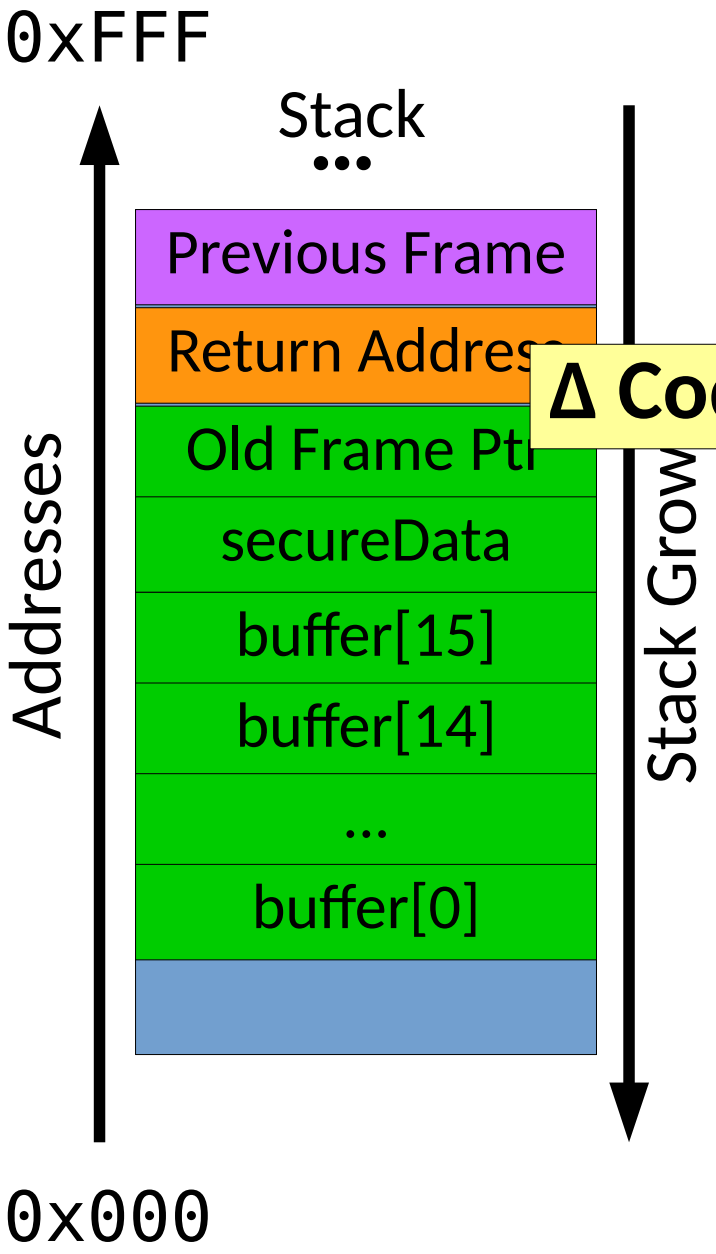
# Control Flow Hijacking




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

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

# Control Flow Hijacking

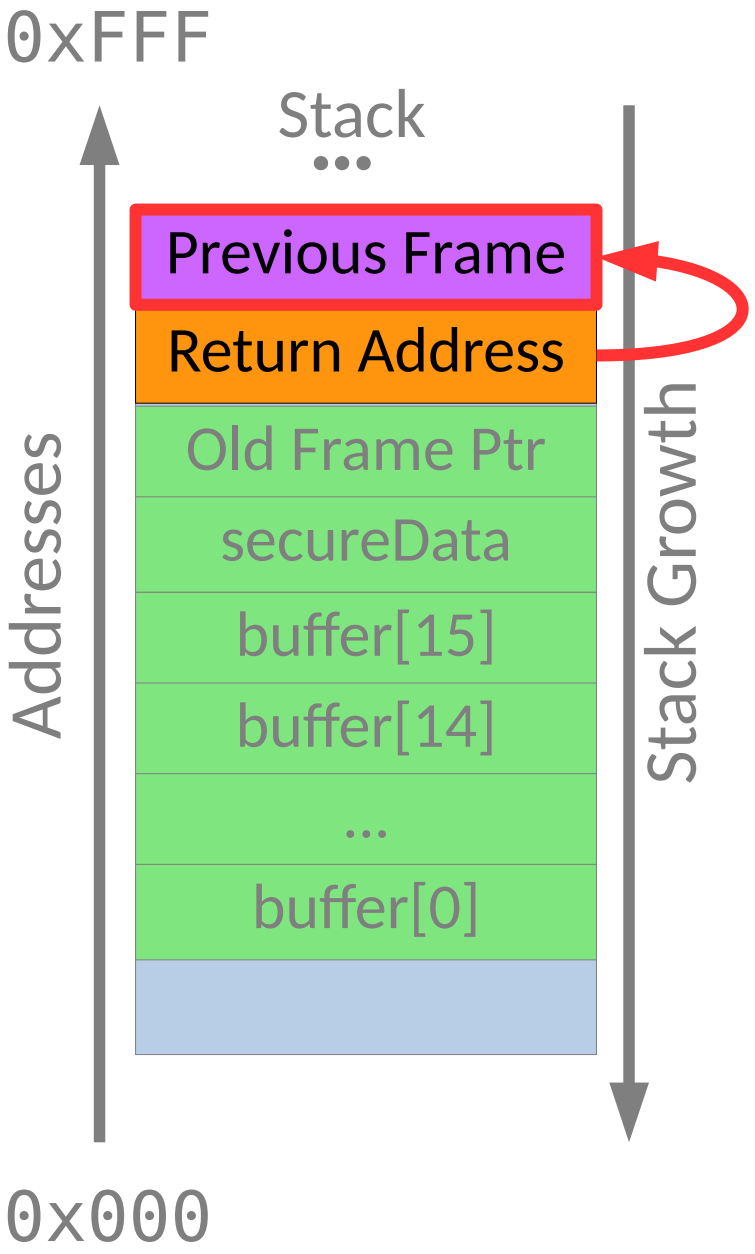



**Δ Code ***

```
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)”

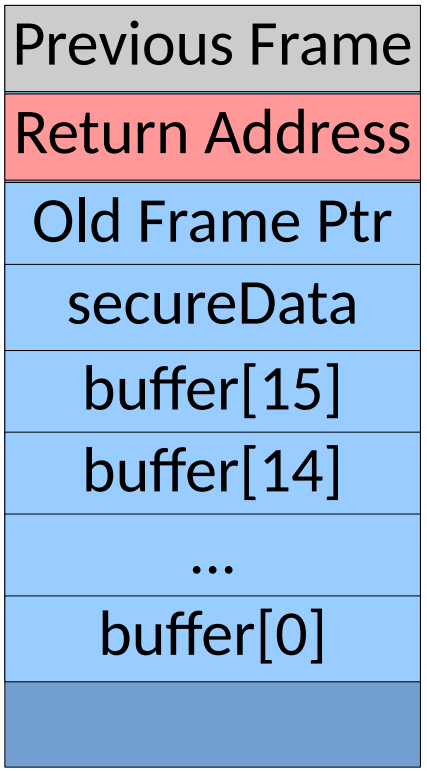
On return, we'll execute the shell code

# Control Flow Hijacking

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

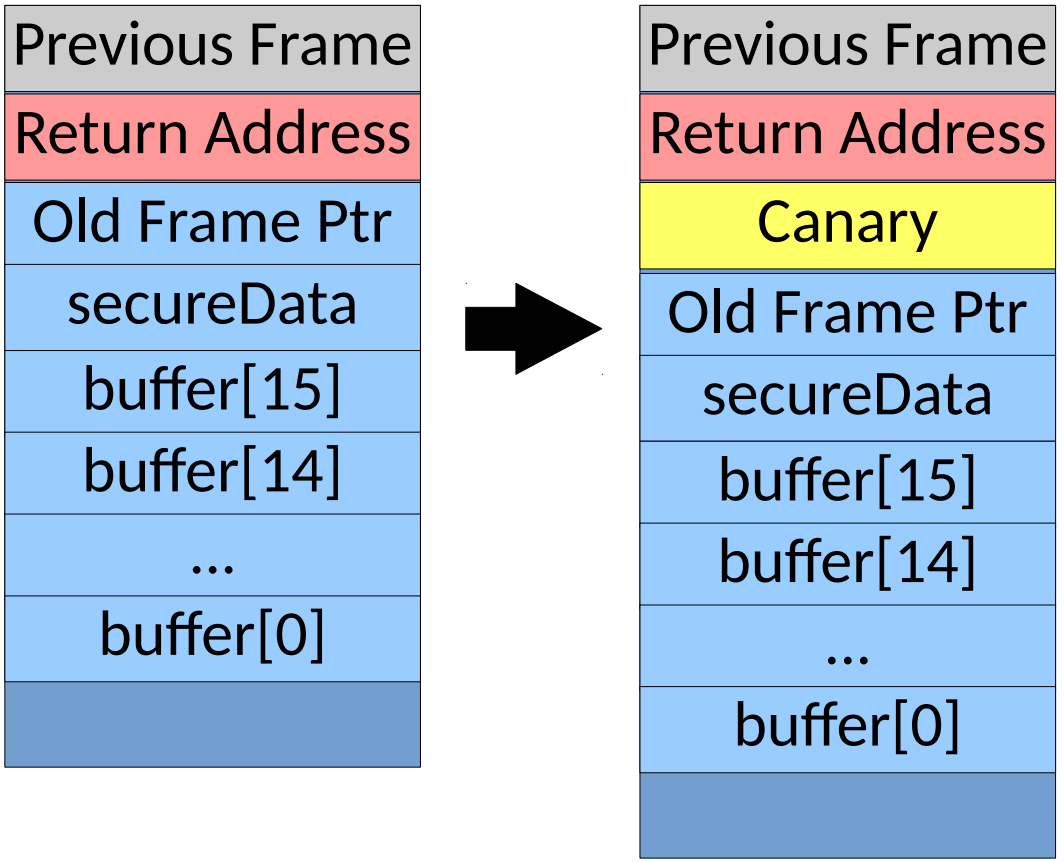
# Control Flow Hijacking

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



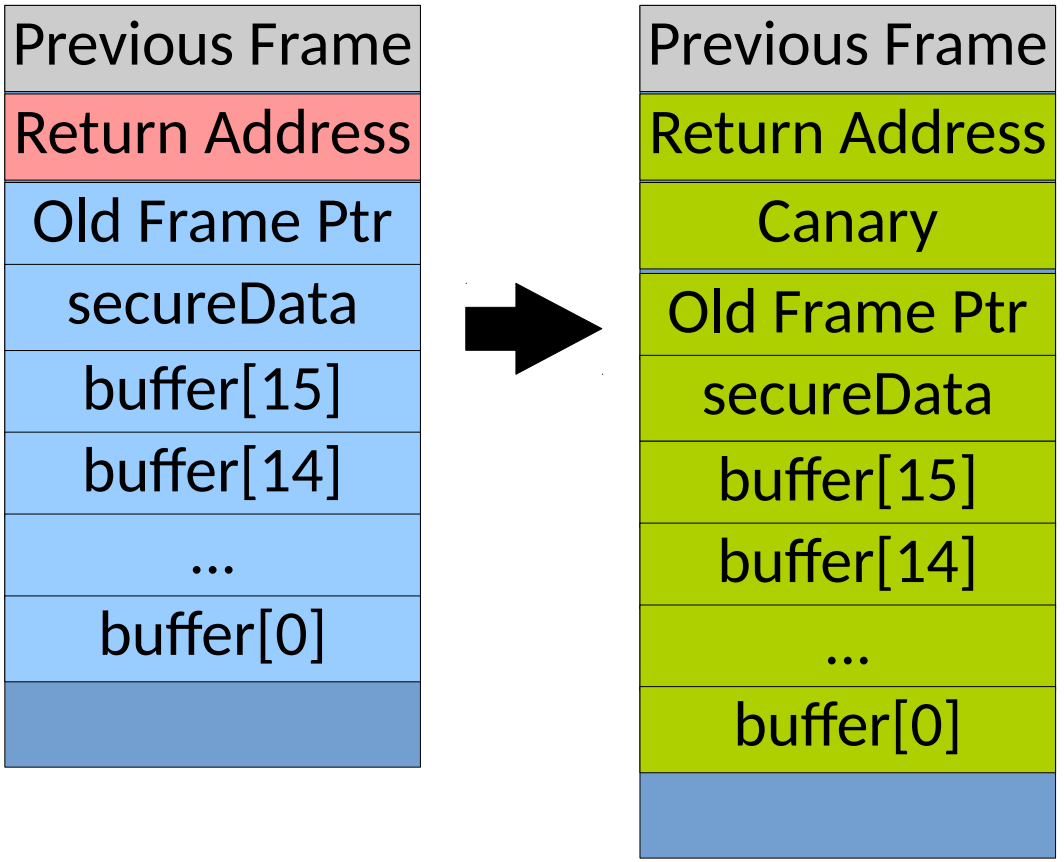
# Control Flow Hijacking

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



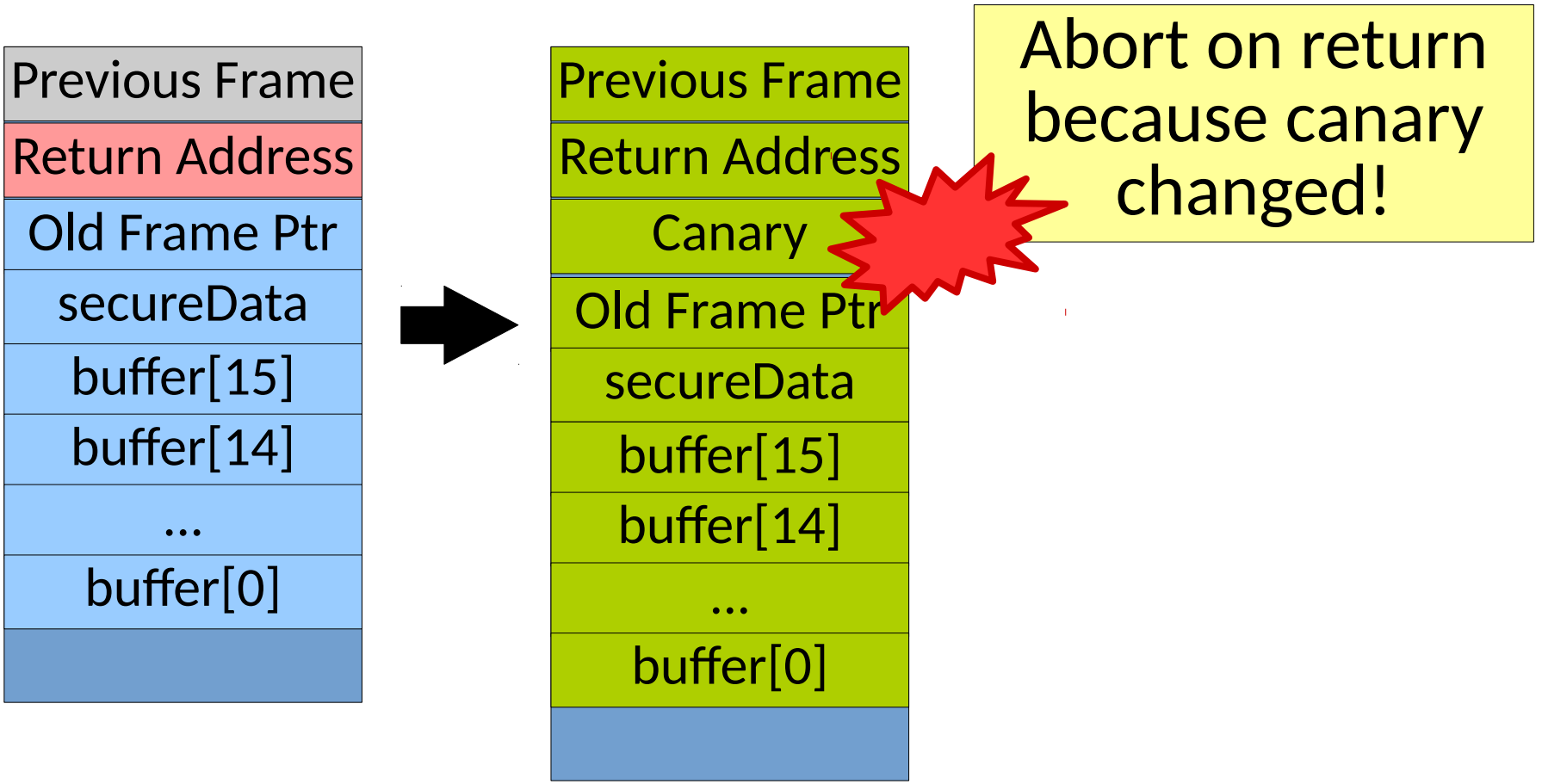
# Control Flow Hijacking

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



# Control Flow Hijacking

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

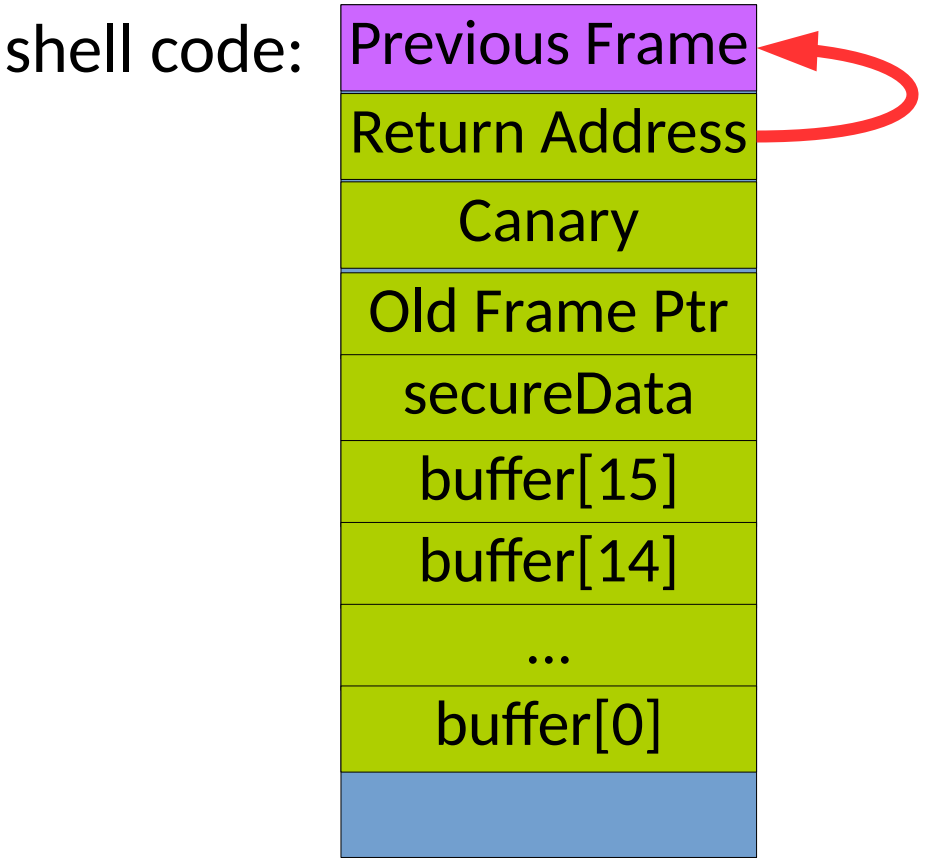


# Control Flow Hijacking

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

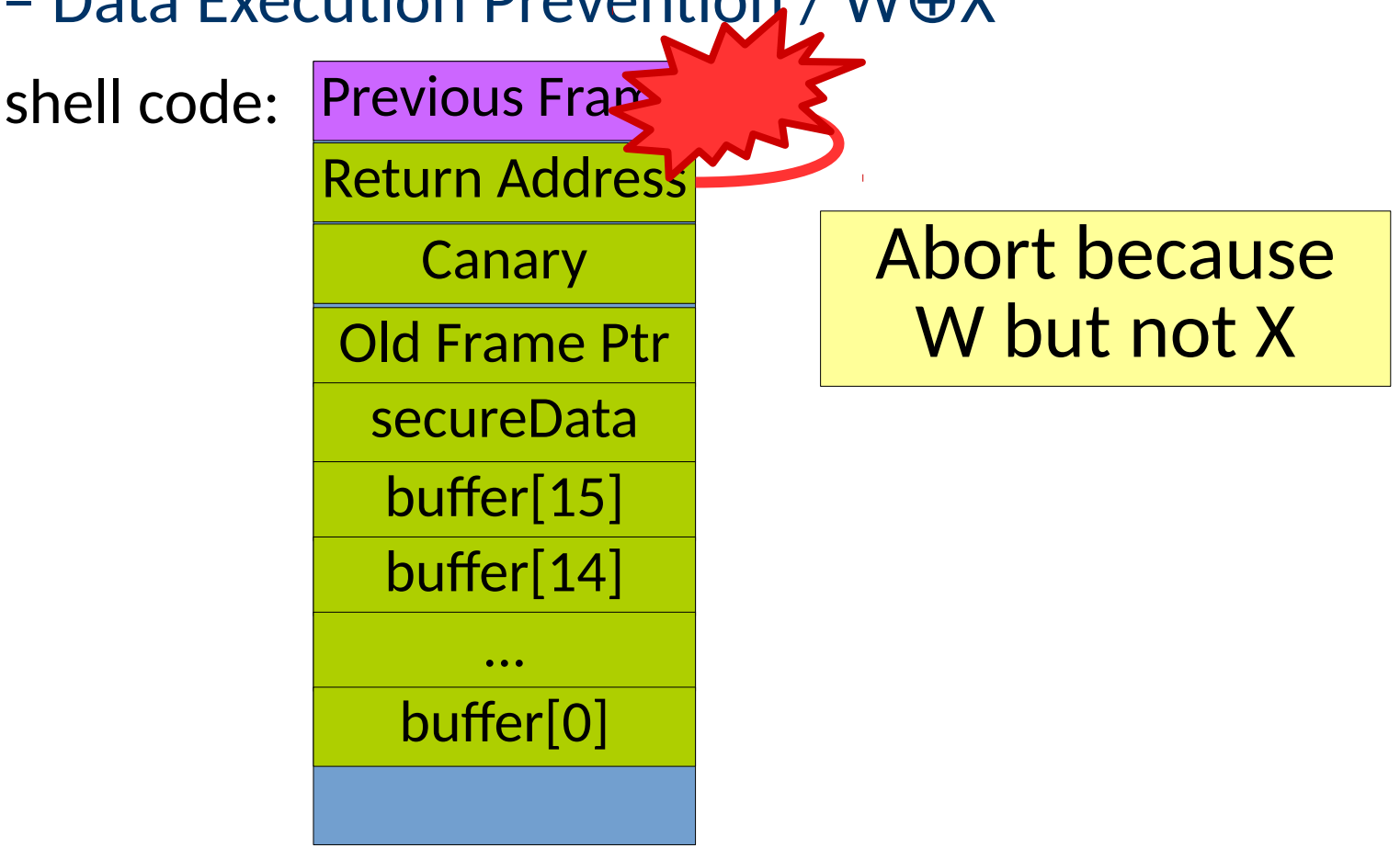
# Control Flow Hijacking

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



# Control Flow Hijacking

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



# Control Flow Hijacking

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

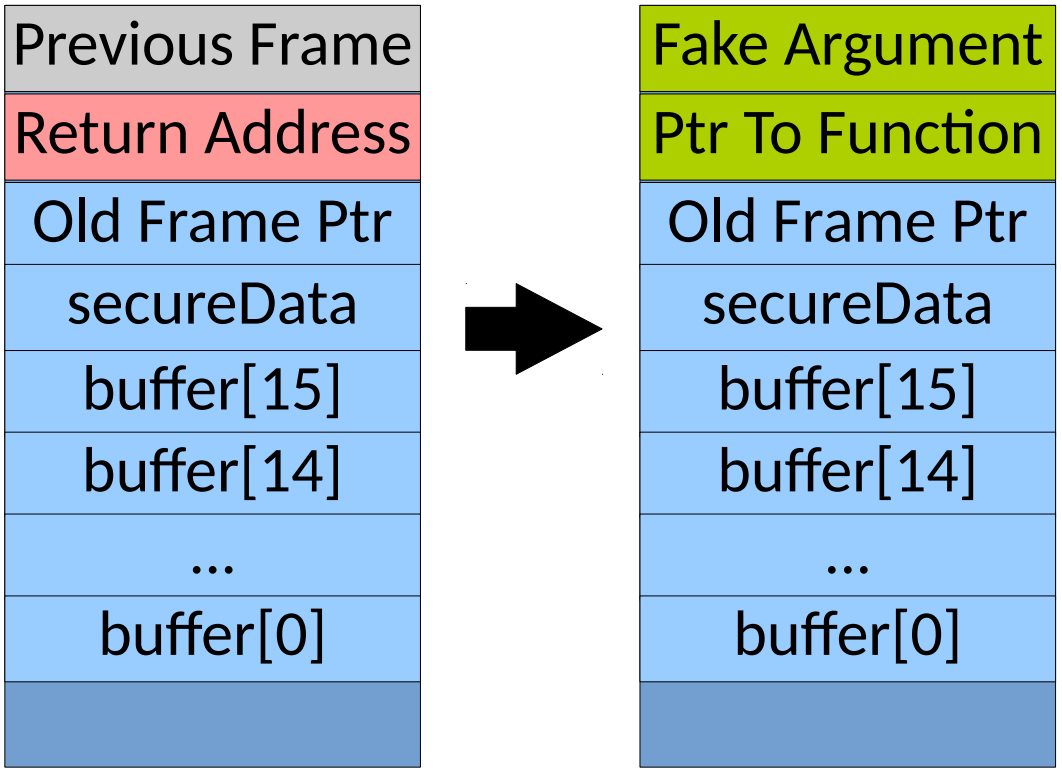
But these are still easily bypassed!

# Return to libc Attacks

- Reuse existing code to bypass W⊕X

# Return to libc Attacks

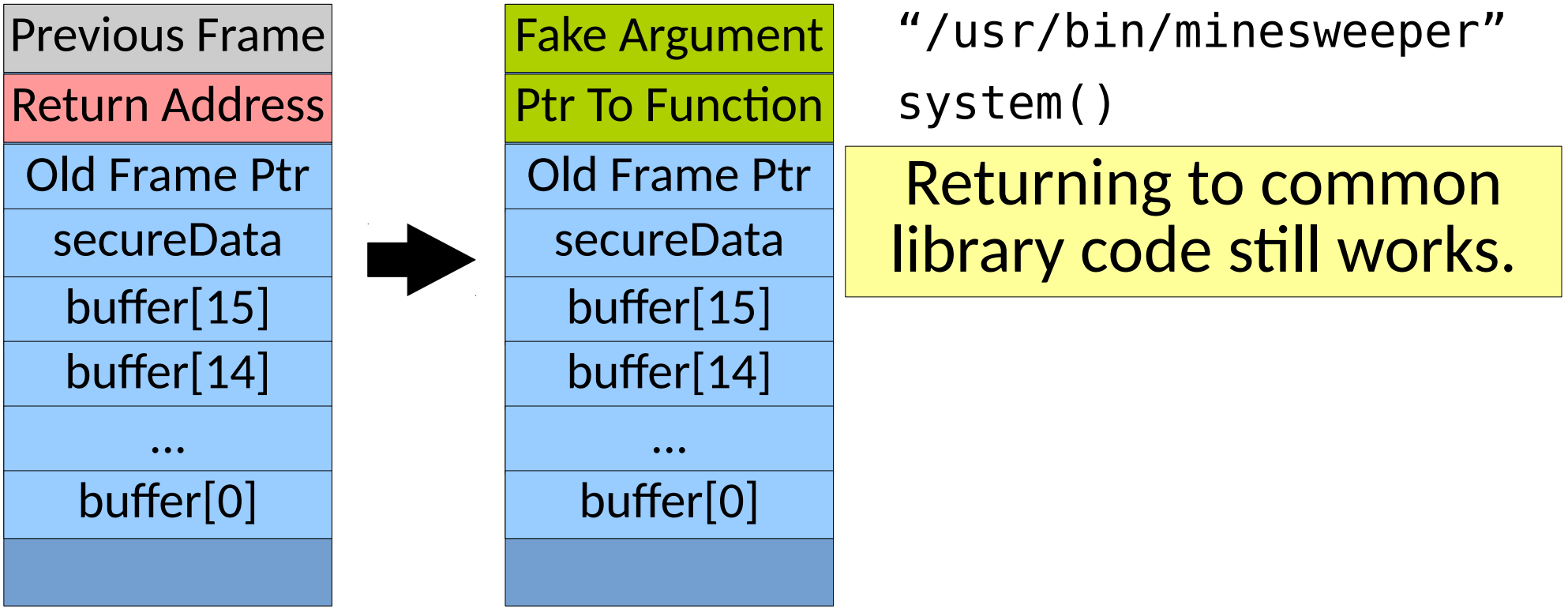
- Reuse existing code to bypass W⊕X




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

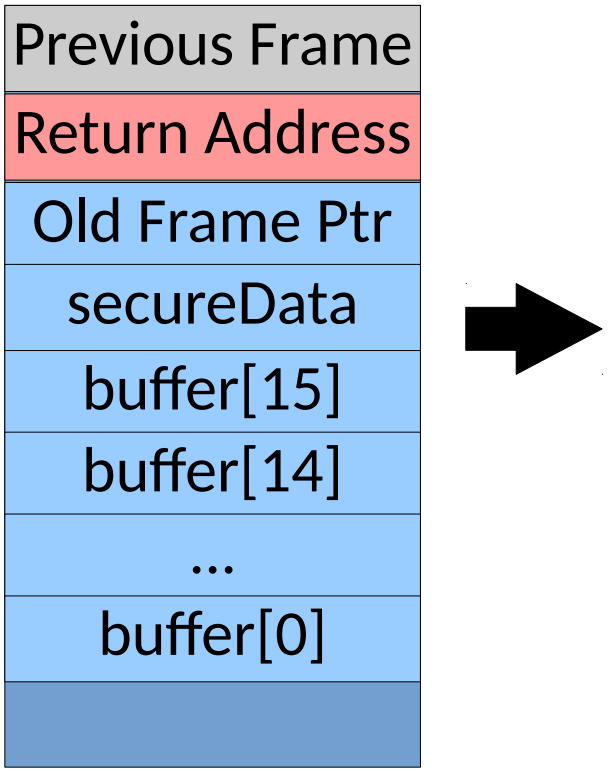
# Return to libc Attacks

- Reuse existing code to bypass W⊕X

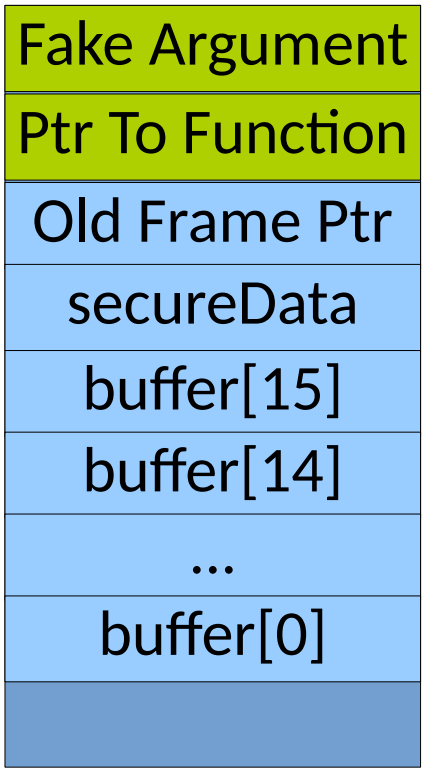


# Return to libc Attacks

- Reuse existing code to bypass W⊕X






`"/usr/bin/minesweeper"`

`system()`

Returning to common library code still works.

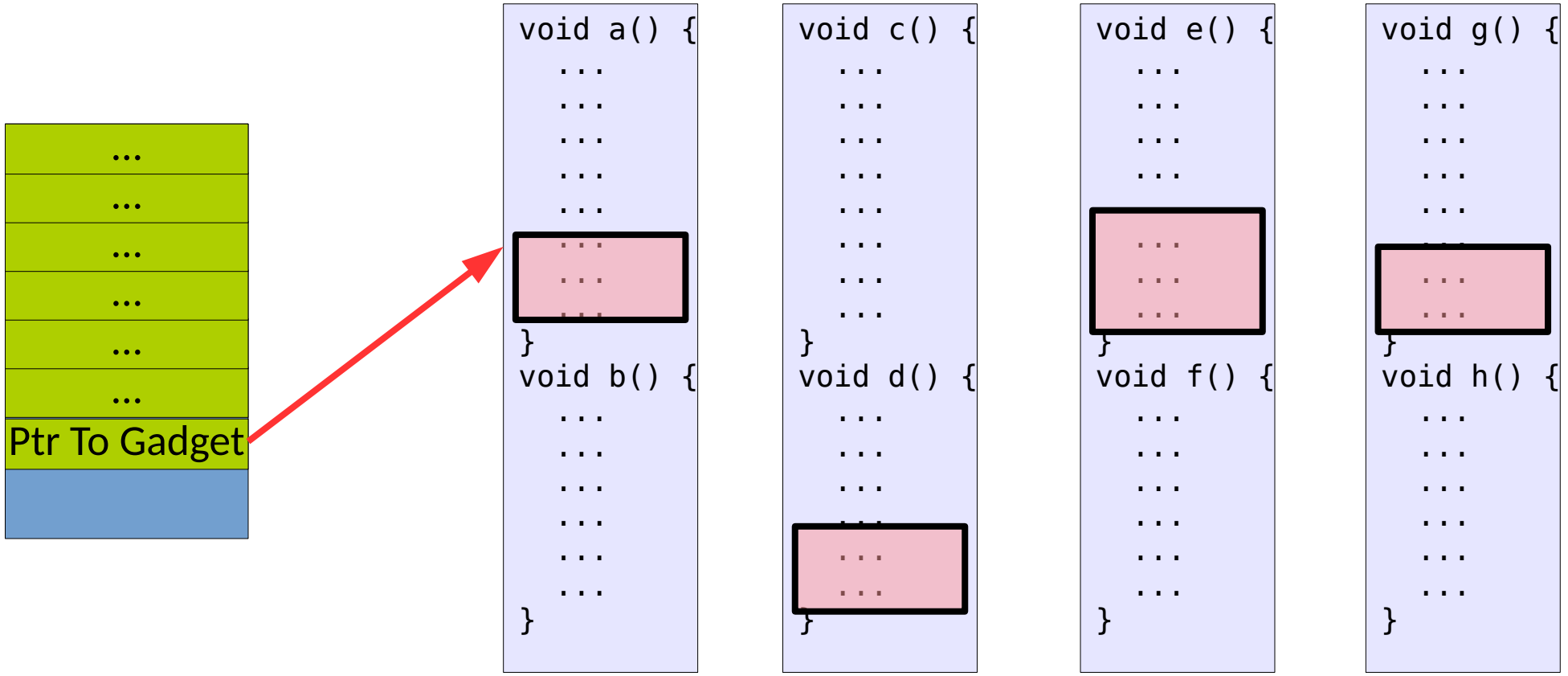
Even construct new functions piece by piece…

# Return to libc Attacks

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

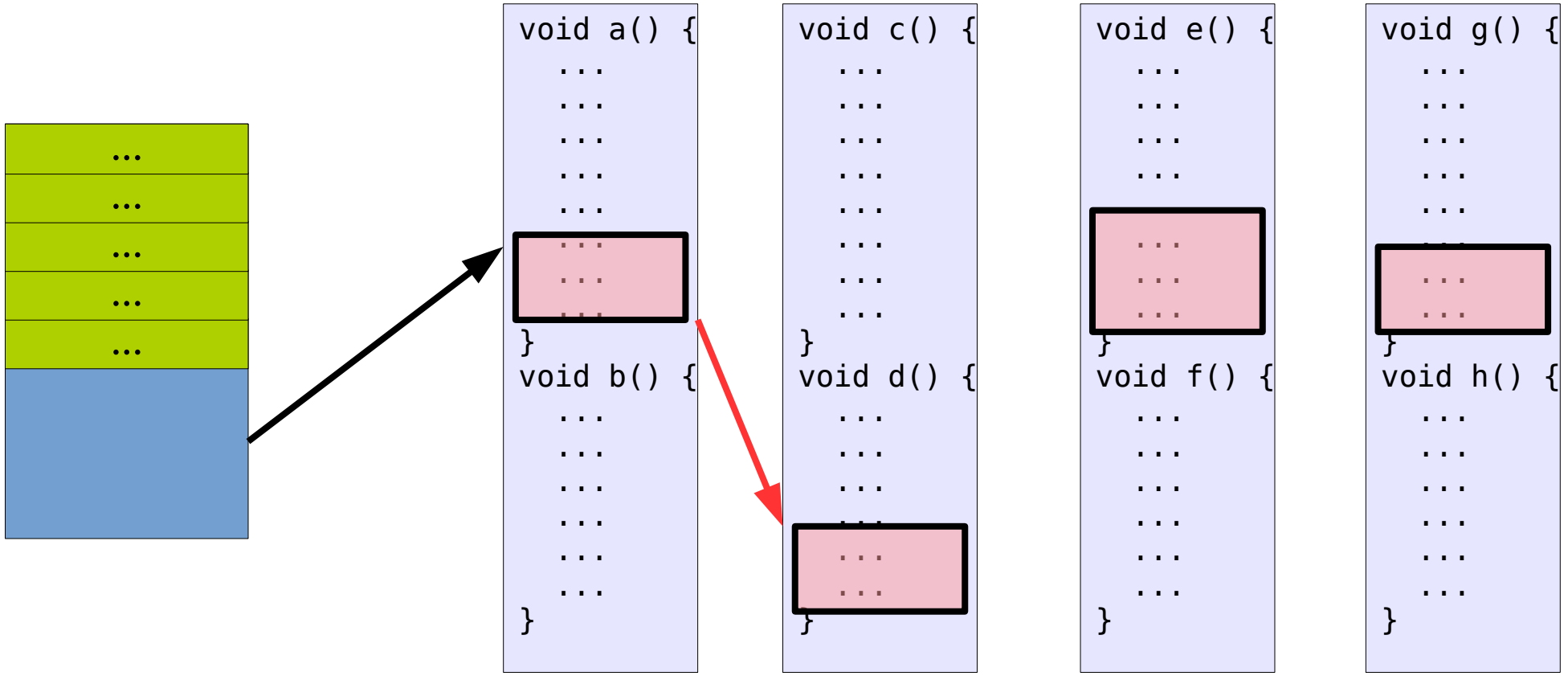
# Return to libc Attacks

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



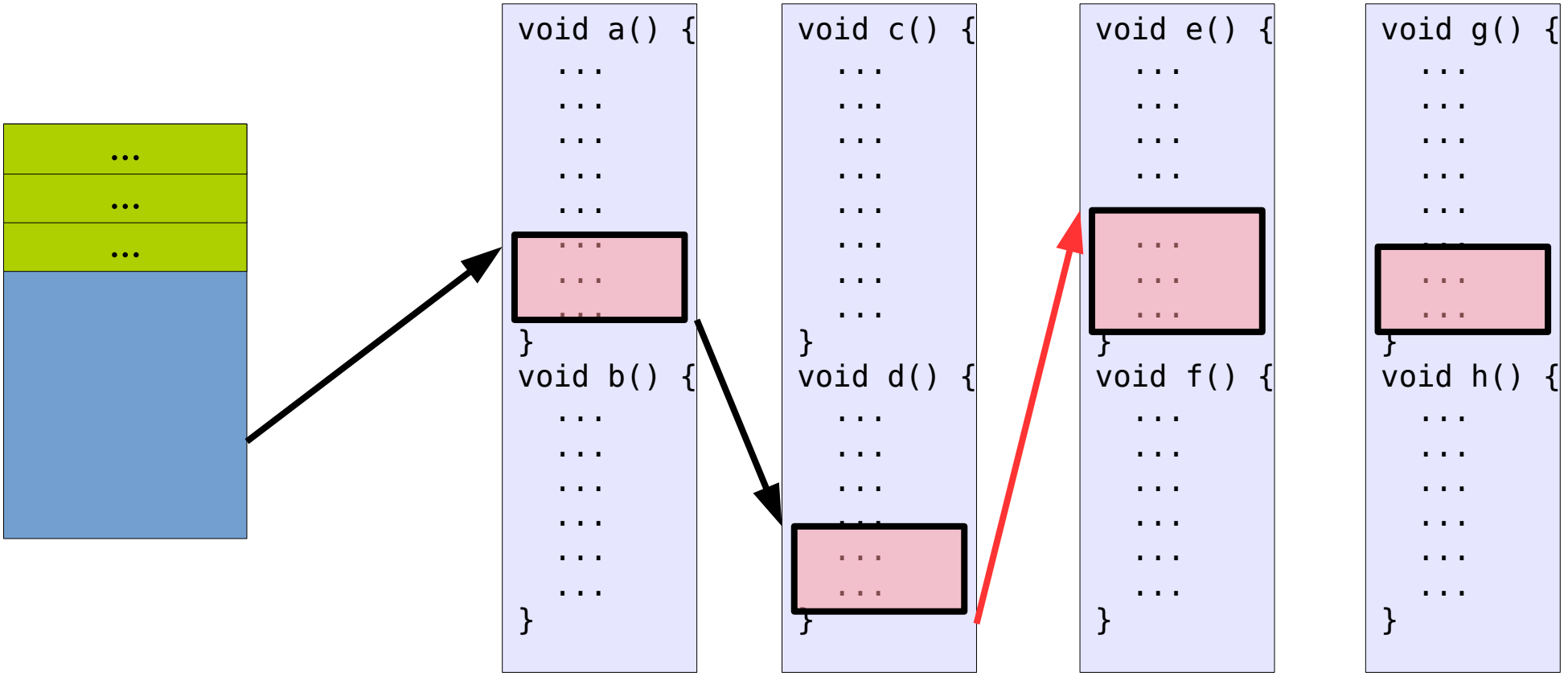
# Return to libc Attacks

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



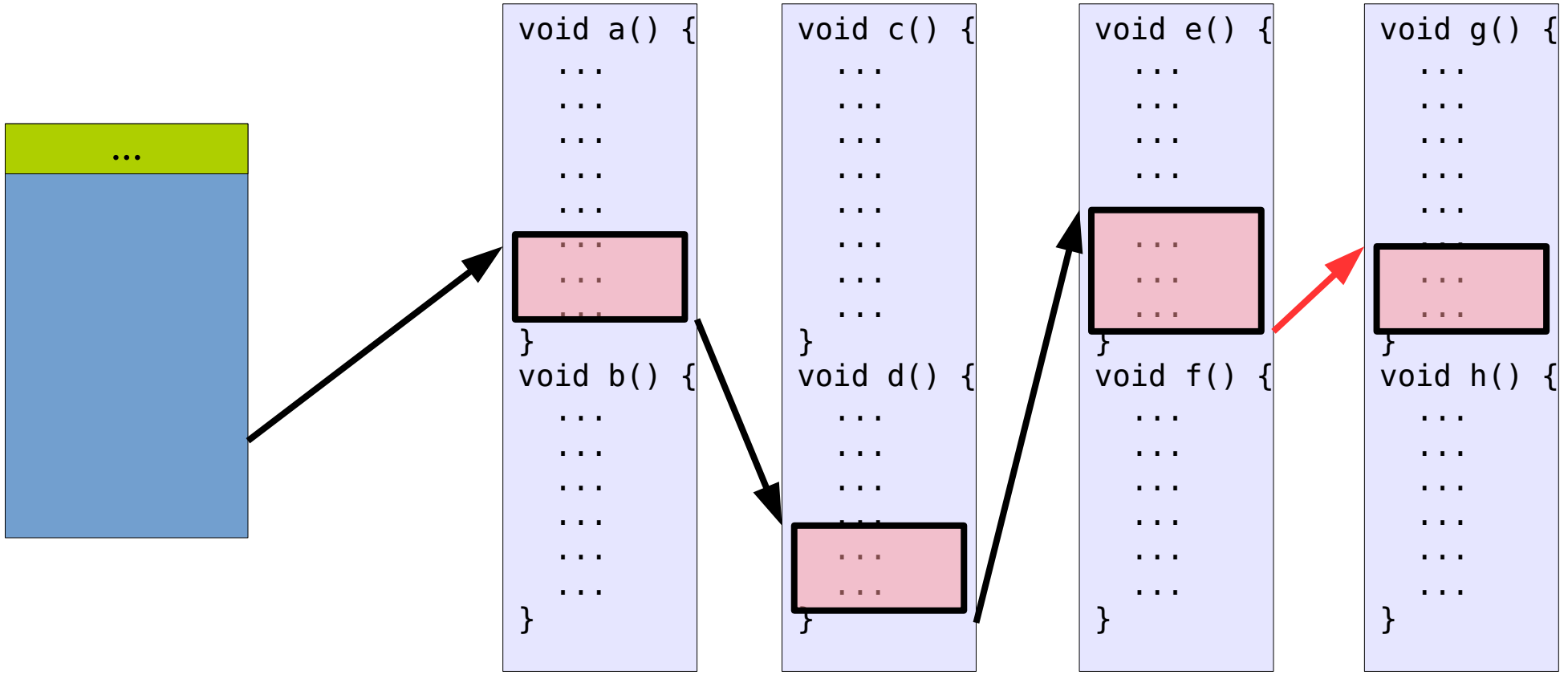
# Return to libc Attacks

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



# Return to libc Attacks

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

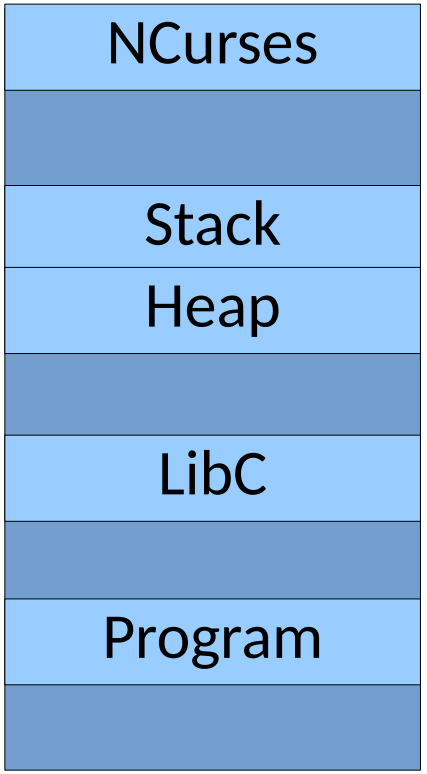


# ASLR

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

# ASLR

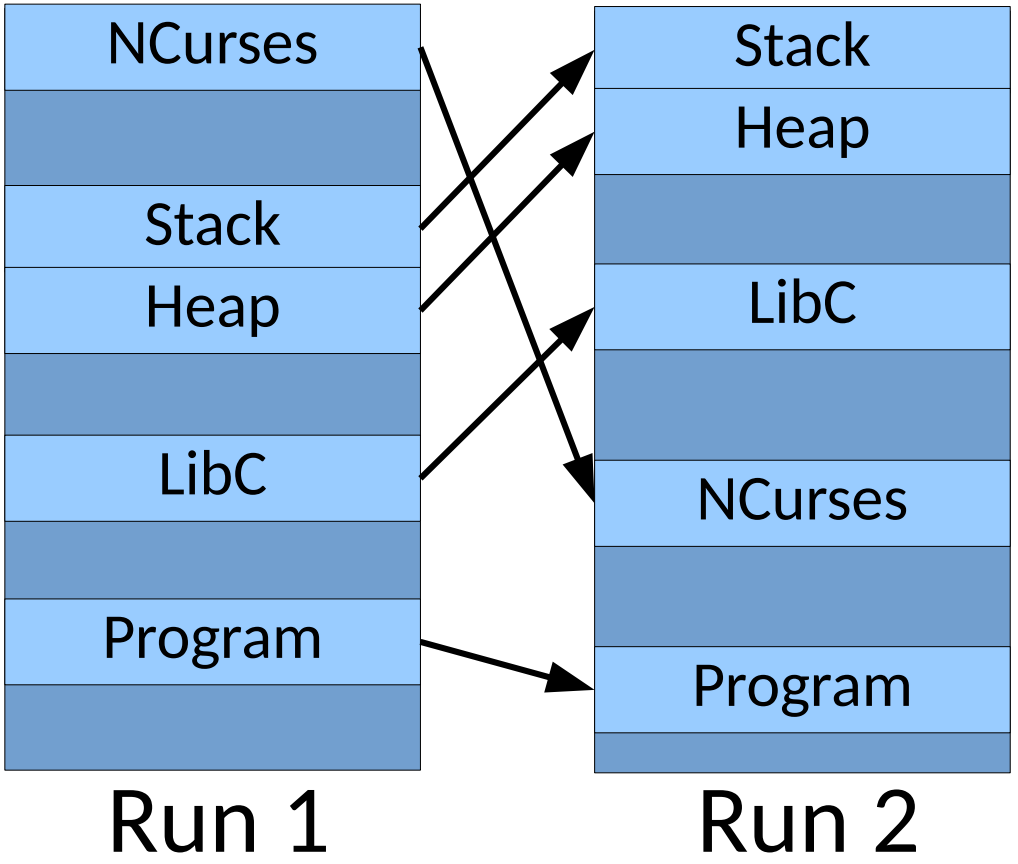
- Address Space Layout Randomization
  - You can't use it if you can't find it!




Run 1

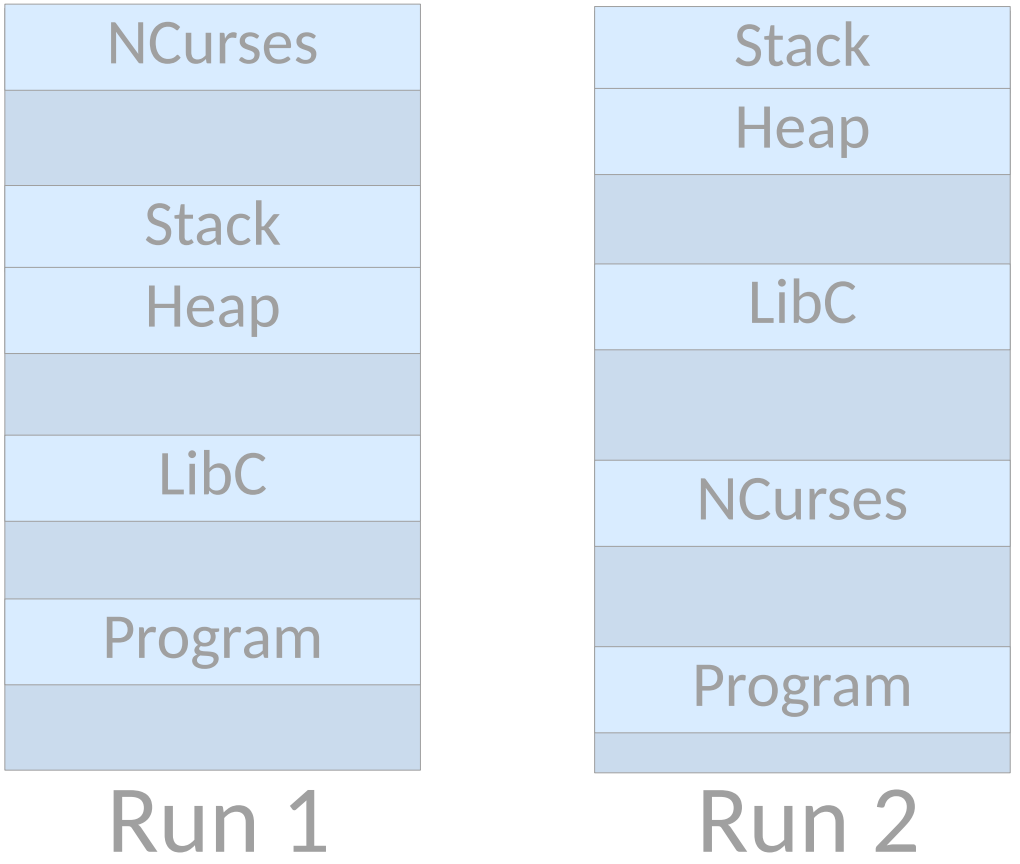
# ASLR

- 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

# Control Flow Integrity

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

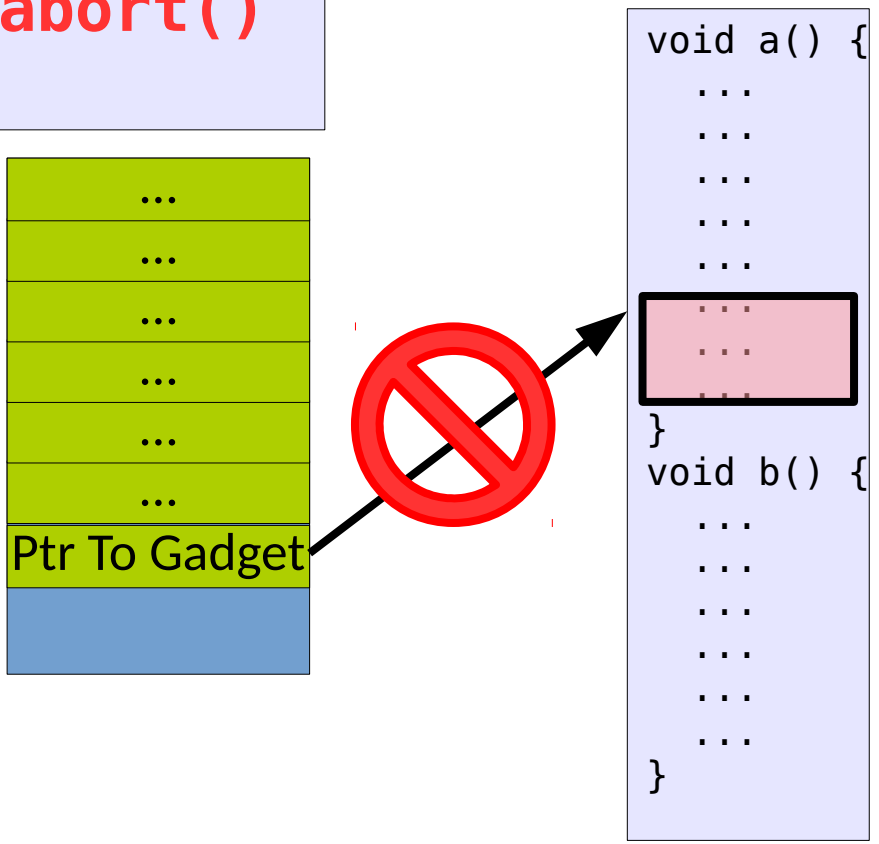
```
foo = ...

foo();
```

# Control Flow Integrity

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

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

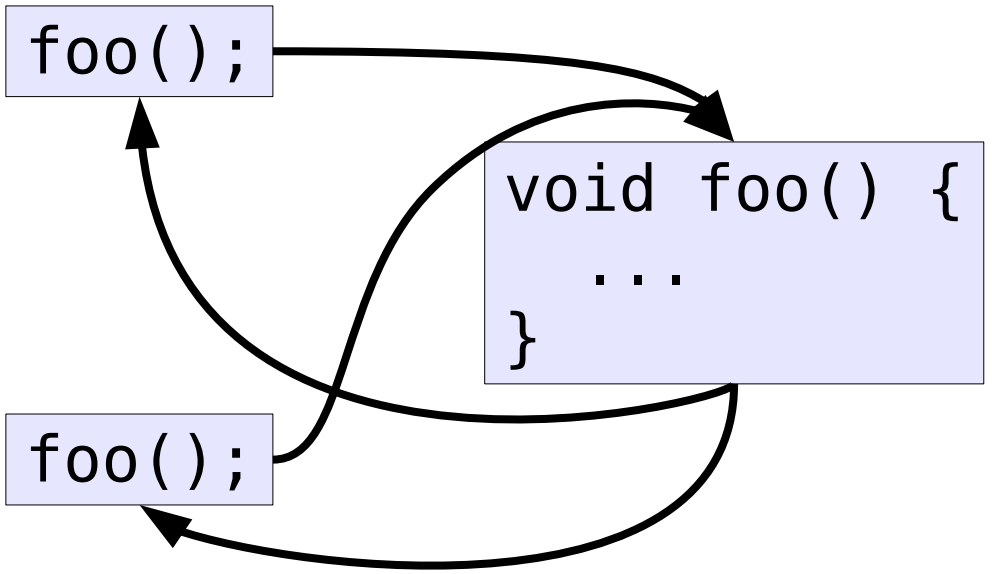


# Control Flow Integrity

- What problem from context sensitivity reappears for returns?

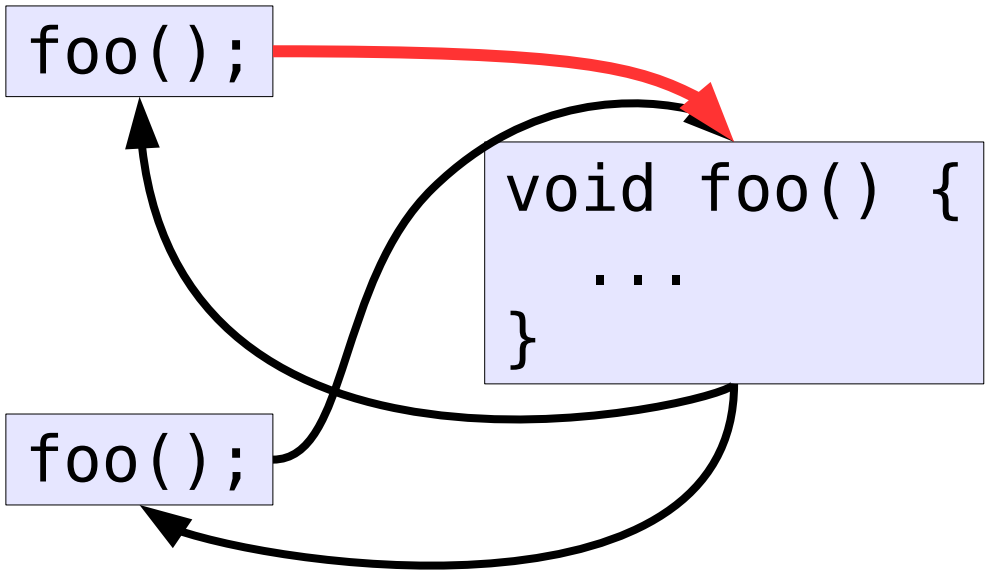
# Control Flow Integrity

- What problem from context sensitivity reappears for returns?



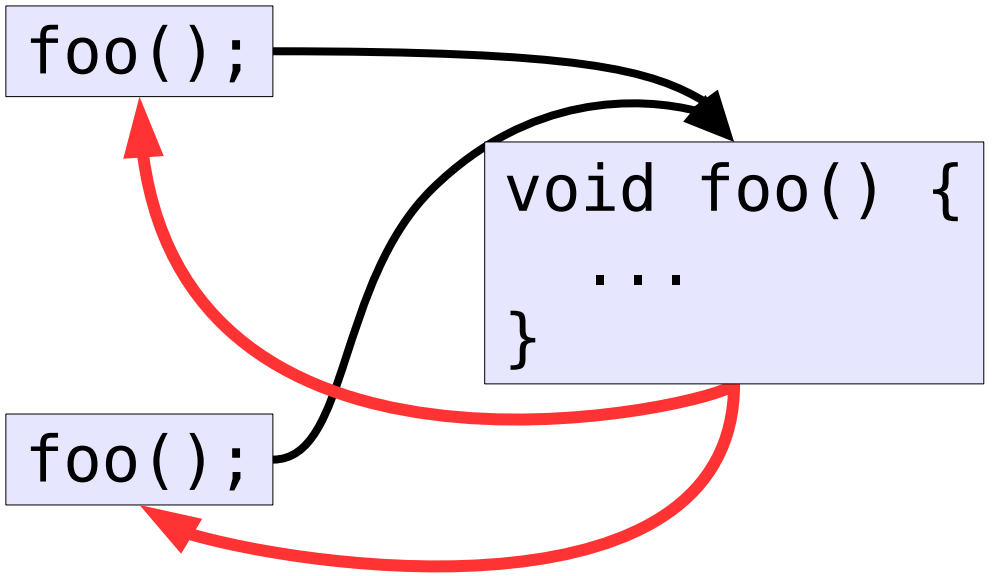
# Control Flow Integrity

- What problem from context sensitivity reappears for returns?



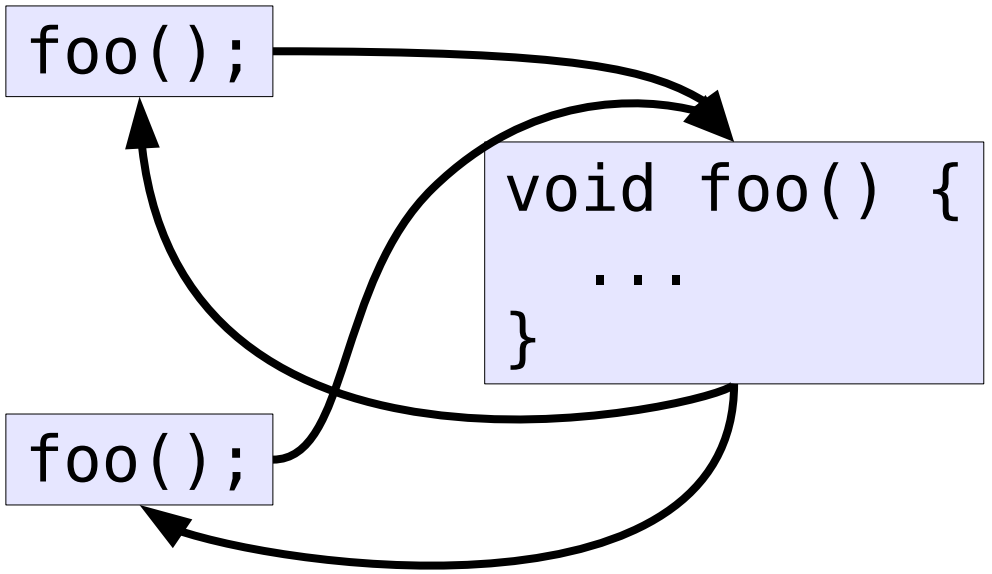
# Control Flow Integrity

- What problem from context sensitivity reappears for returns?



# Control Flow Integrity

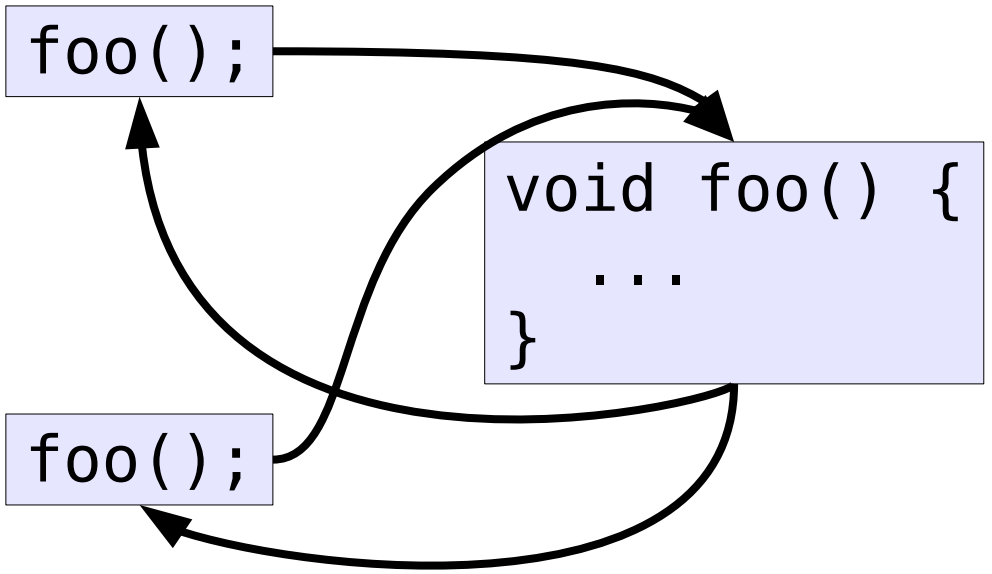
- What problem from context sensitivity reappears for returns?




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

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

# Control Flow *Bending*

- Suppose we have perfect CFI; how good is it?

# Control Flow Bending

- Suppose we have perfect CFI; how good is it?
  - Can we skip using stacks?

# Control Flow Bending

- Suppose we have perfect CFI; how good is it?
  - Can we skip using stacks?

**No!**

# Control Flow Bending

- Suppose we have perfect CFI; how good is it?
  - Can we skip using stacks?

Why might CFI offer porous defenses?

# 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

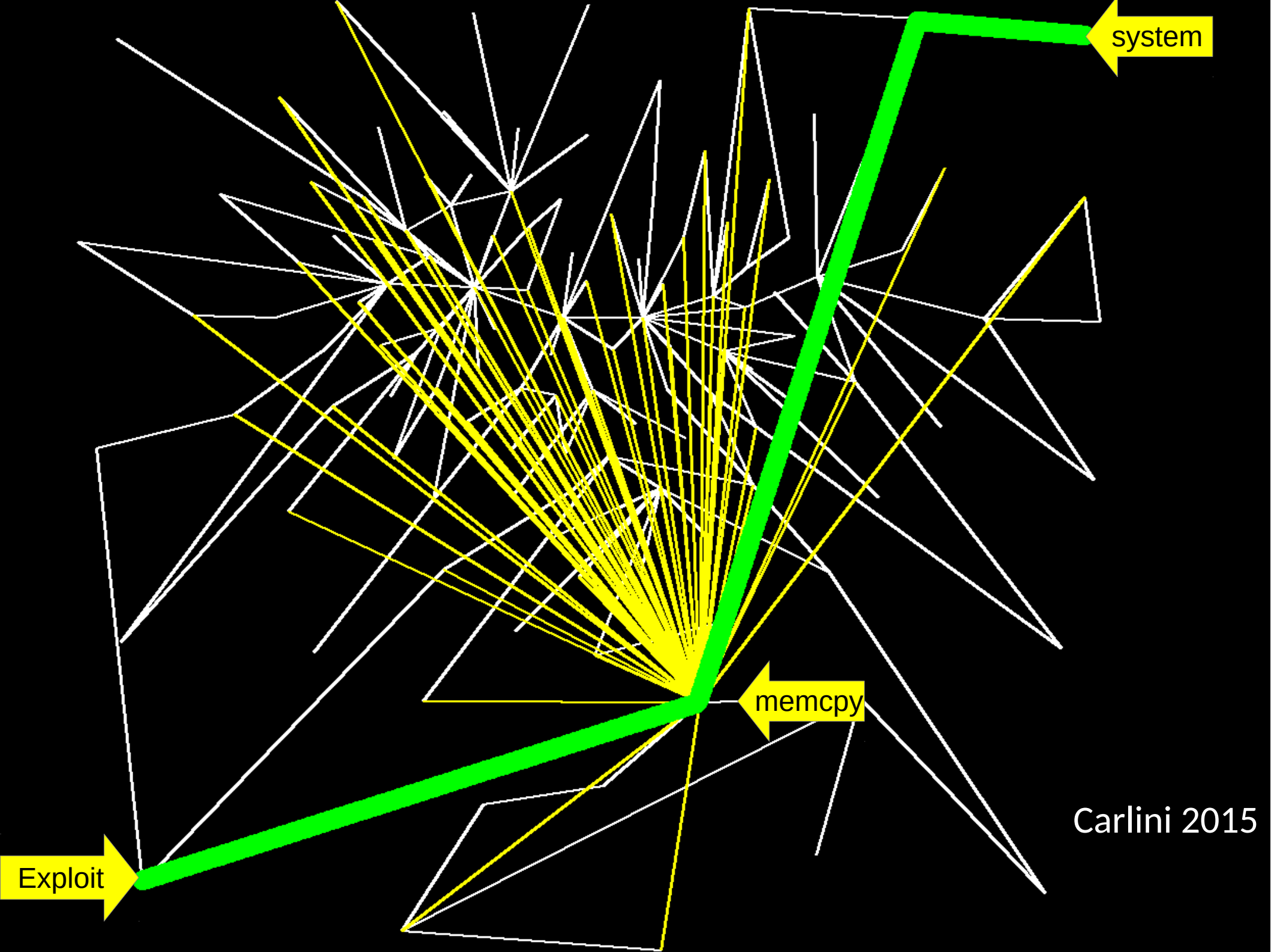
If fully precise CFI is broken, then CFI is broken.

# Dispatcher Functions

- Some key functions open up substantial opportunities, e.g. memcpy

# Dispatcher Functions

- Some key functions open up substantial opportunities, e.g. memcpy
- Any function that calls them becomes an attack surface


system
memcpy
Exploit
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 advanced return to libc!

# What Does CFI+Shadow Stacks Give?___

- No longer able to do ROP
- Still able to do advanced return to libc!
- Worse: **`printf`** alone provides a Turing complete attack surface

# Ongoing Directions

- What if violating CFI is not even necessary?

# Ongoing Directions

- What if violating CFI is not even necessary?
- Data Oriented Programming (DOP)

# Ongoing Directions

- What if violating CFI is not even necessary?
- Data Oriented Programming (DOP)
  - Turn a program into an interpreter for a malicious input

# Ongoing Directions

- What if violating CFI is not even necessary?

- Data Oriented Programming (DOP)
  - Turn a program into an interpreter for a malicious input
  - Without modifying the control flow at all.

# Ongoing Directions

- What if violating CFI is not even necessary?
- Data Oriented Programming (DOP)
  - Turn a program into an interpreter for a malicious input
  - Without modifying the control flow at all.
- It's an emerging vector that needs new mitigation techniques.

# Further Defenses

- Sandboxing / isolation

# Further Defenses

- Sandboxing / isolation
- Architectural support for security (e.g. Intel SGX)

# Further Defenses

- Sandboxing / isolation
- Architectural support for security (e.g. Intel SGX)
- **`pledge()`** based access control (OpenBSD)

# Further Defenses

- Sandboxing / isolation
- Architectural support for security (e.g. Intel SGX)
- `pledge()` based access control (OpenBSD)
- Code randomization
- …

# Summary

- Security is a *critical* concern in modern software

# Summary

- Security is a *critical* concern in modern software
- There is an ongoing struggle between attackers & defenders

# Summary

- Security is a *critical* concern in modern software
- There is an ongoing struggle between attackers & defenders
- Memory safety violations provide a common vector of attack