CSE467: Computer Security

10. Type Confusion & Control Flow Integrity

Seongil Wi

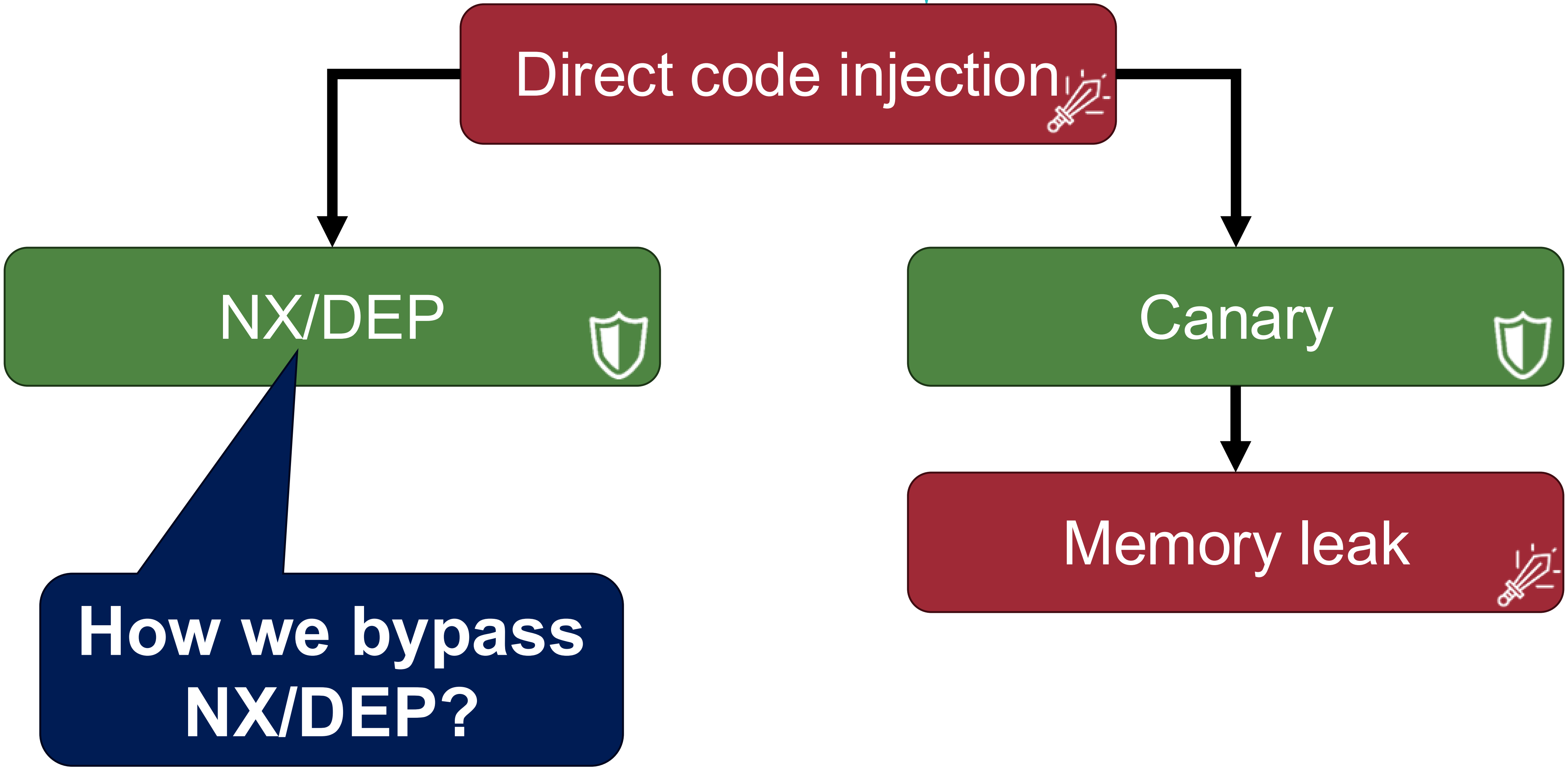
Department of Computer Science and Engineering

The slide is based on Prof. Sang Kil Cha's lecture slide

HW2

- Write a critique for the two papers:
  - WYSINWYX: What you see is not what you eXecute, ***TOPLAS 2005***
  - The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), ***CCS 20107***
- Due: November 11, 11:59PM

Recap: Bypassing NX/DEP



Recap: Bypassing DEP

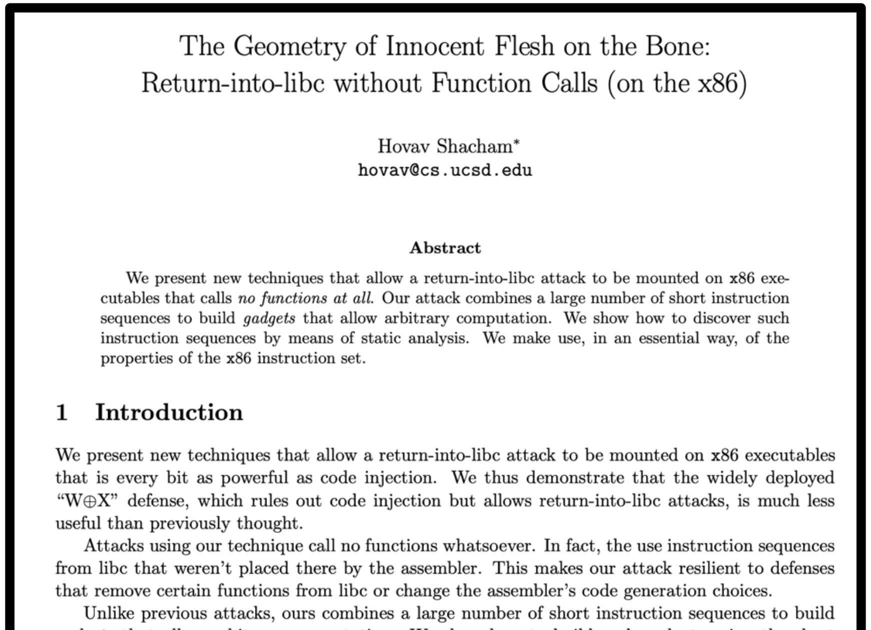
- Return-to-stack exploit is disabled
- But, we can still jump to an arbitrary address of ***existing code*** **(=** ***Code Reuse Attack*****)**

Recap: ROP (Return-oriented Programming)

Generalized Code Reuse Attack

Formally introduced by Hovav in CCS 2007

“The Geometry of Innocent Flesh on the Bone: Return-to-libc without Function Calls (on the x86)”



The Geometry of Innocent Flesh on the Bone:
Return-into-libc without Function Calls (on the x86)

Hovav Shacham*
hovav@cs.ucsd.edu

Abstract

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls *no functions at all*. Our attack combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to discover such instruction sequences by means of static analysis. We make use, in an essential way, of the properties of the x86 instruction set.

1 Introduction

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that is every bit as powerful as code injection. We thus demonstrate that the widely deployed “W⊕X” defense, which rules out code injection but allows return-into-libc attacks, is much less useful than previously thought.

Attacks using our technique call no functions whatsoever. In fact, the use instruction sequences from libc that weren’t placed there by the assembler. This makes our attack resilient to defenses that remove certain functions from libc or change the assembler’s code generation choices.

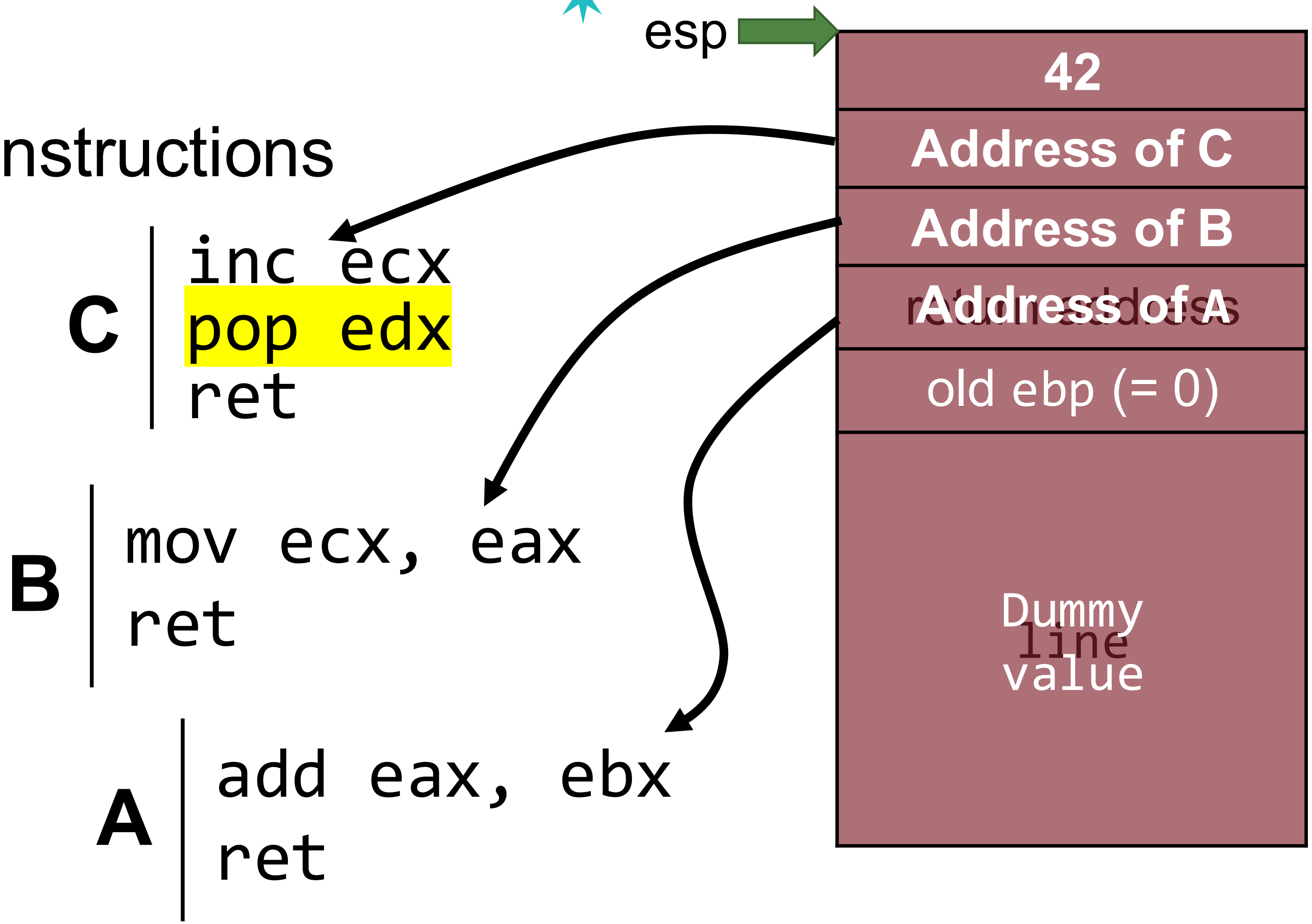
Unlike previous attacks, ours combines a large number of short instruction sequences to build

Recap: Return (ret) Chaining

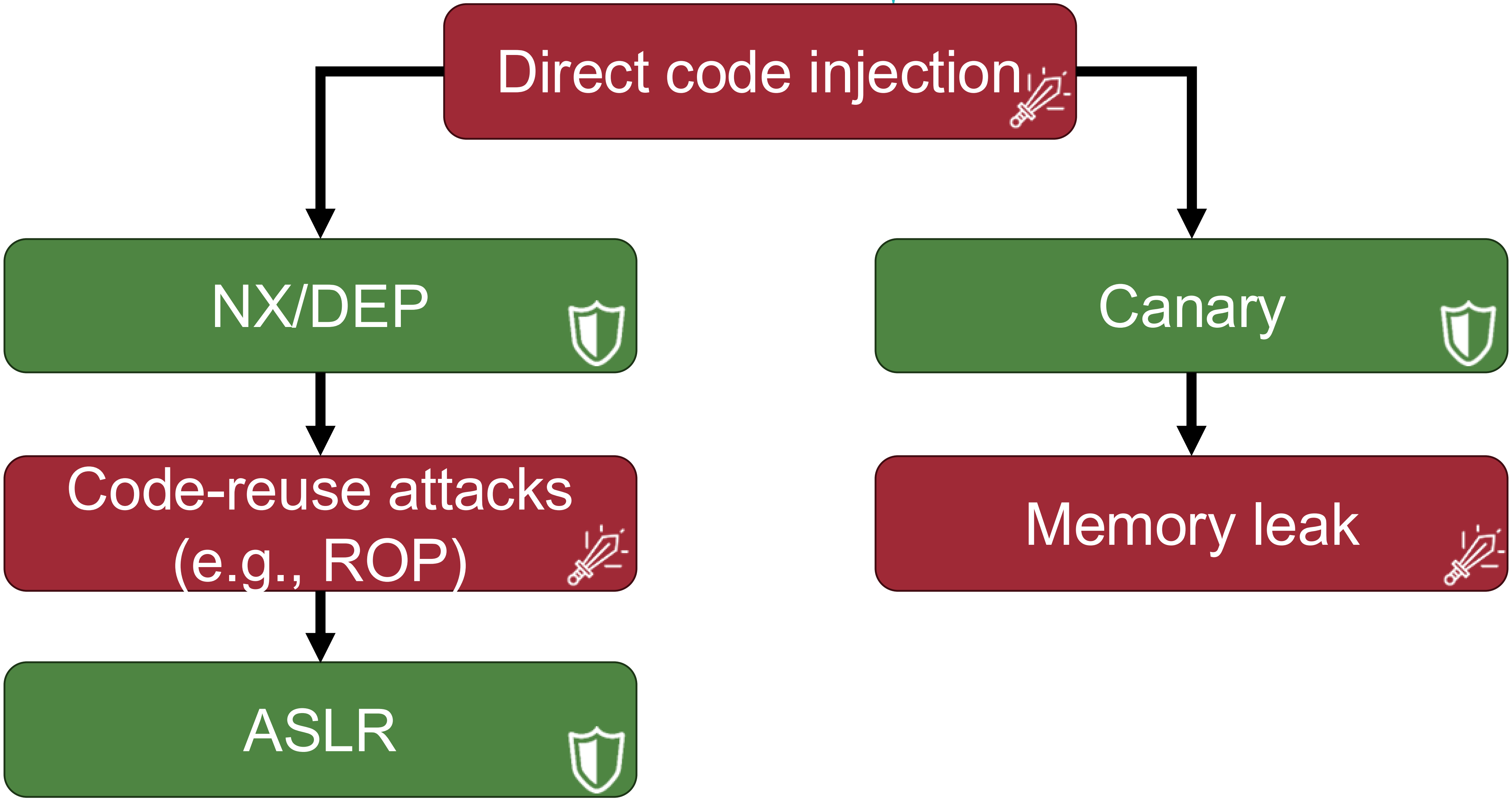
Attacker's goal:

execute following instructions

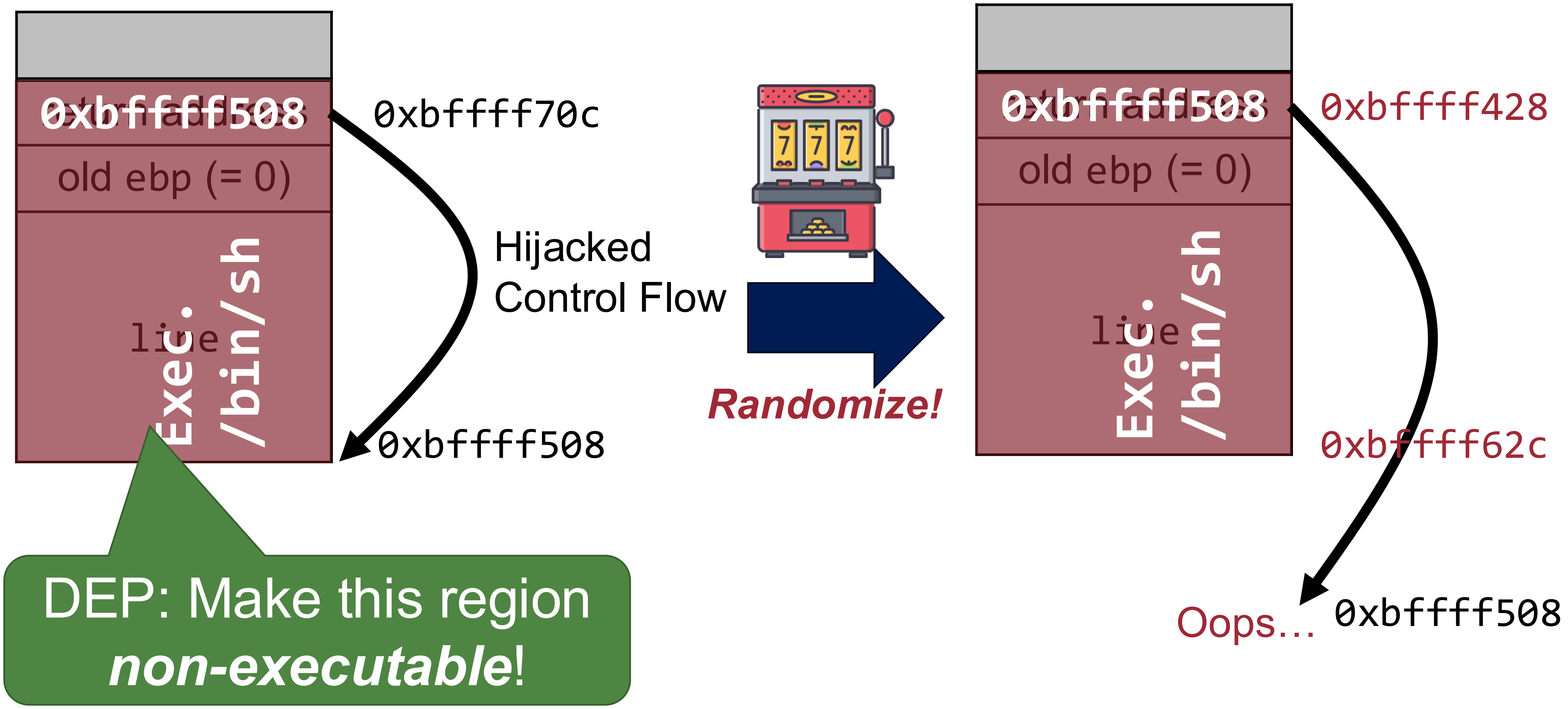
```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```



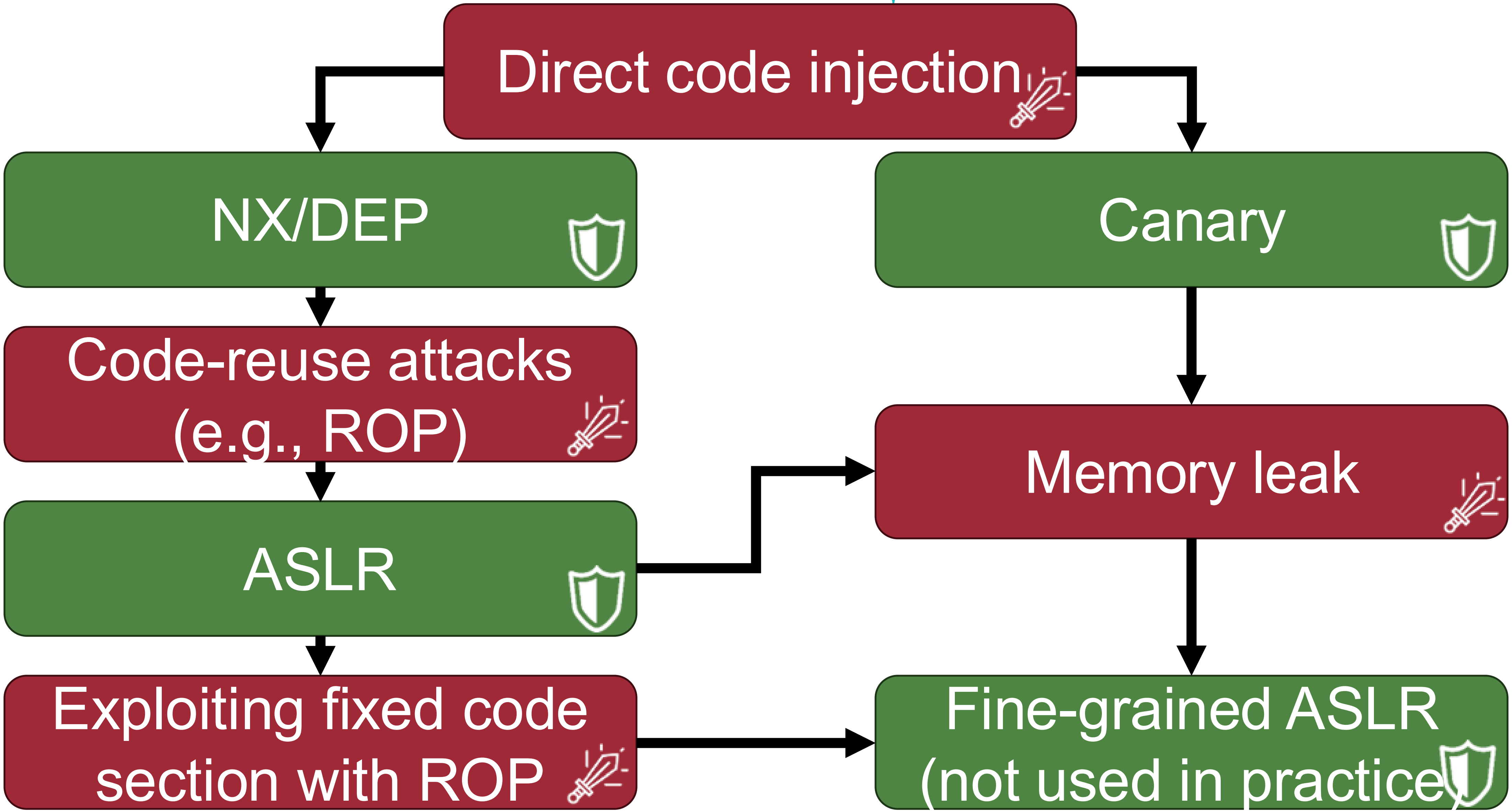
Recap: ASLR


Direct code injection
NX/DEP
Canary
Code-reuse attacks
(e.g., ROP)
Memory leak
ASLR

Recap: ASLR


0xbffff508
old ebp (= 0)
Exec.
/bin/sh
0xbffff70c
0xbffff508
Hijacked
Control Flow
Randomize!
0xbffff508
old ebp (= 0)
Exec.
/bin/sh
0xbffff428
0xbffff62c
Oops…
0xbffff508
DEP: Make this region
non-executable!

Attack / Defense So Far



Memory Corruption So Far

- Buffer overflows

What is another major ***attack vector***
to corrupt memory?

Type Confusion

Type

A classification of data which tells the compiler or interpreter how the programmer intends to use the data

Type Safety

Types prevent unintended errors

```
>>> 1 + "1"

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Type Confusion

- Mistaking a memory location for certain type as a memory for different type

- Type confusion happens when the type-safety is violated

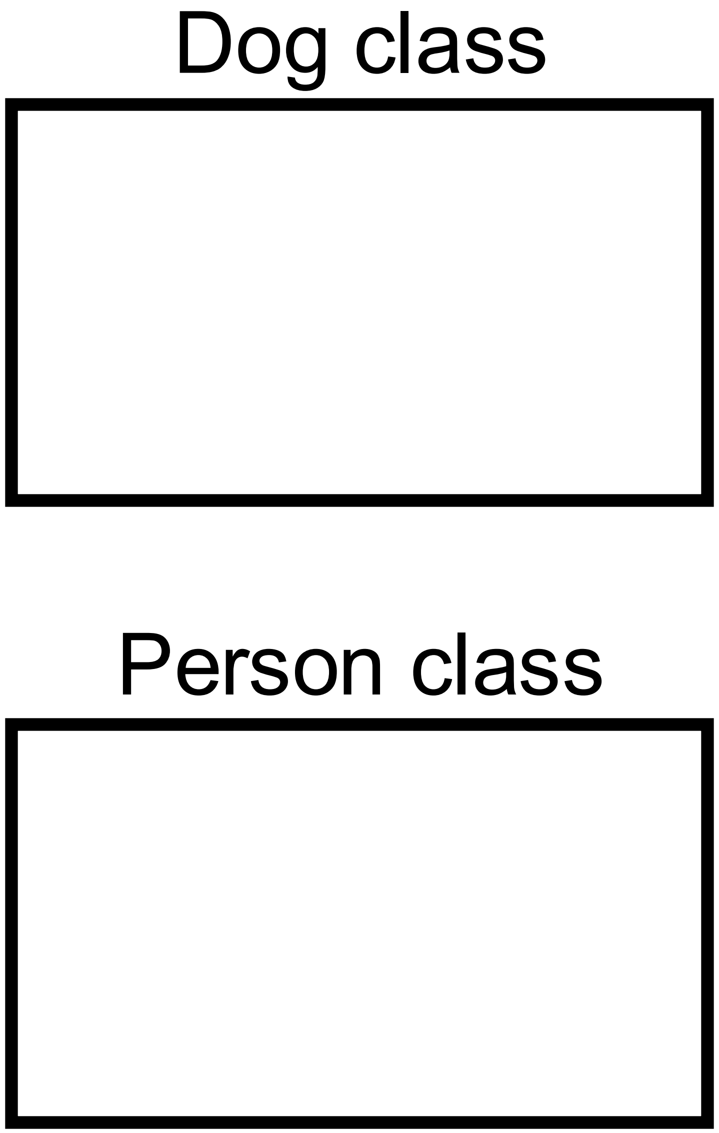
Type Confusion

Normal

```
Dog *d = (Dog*) some_ptr;
d->bark();
```

Abnormal

```
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```

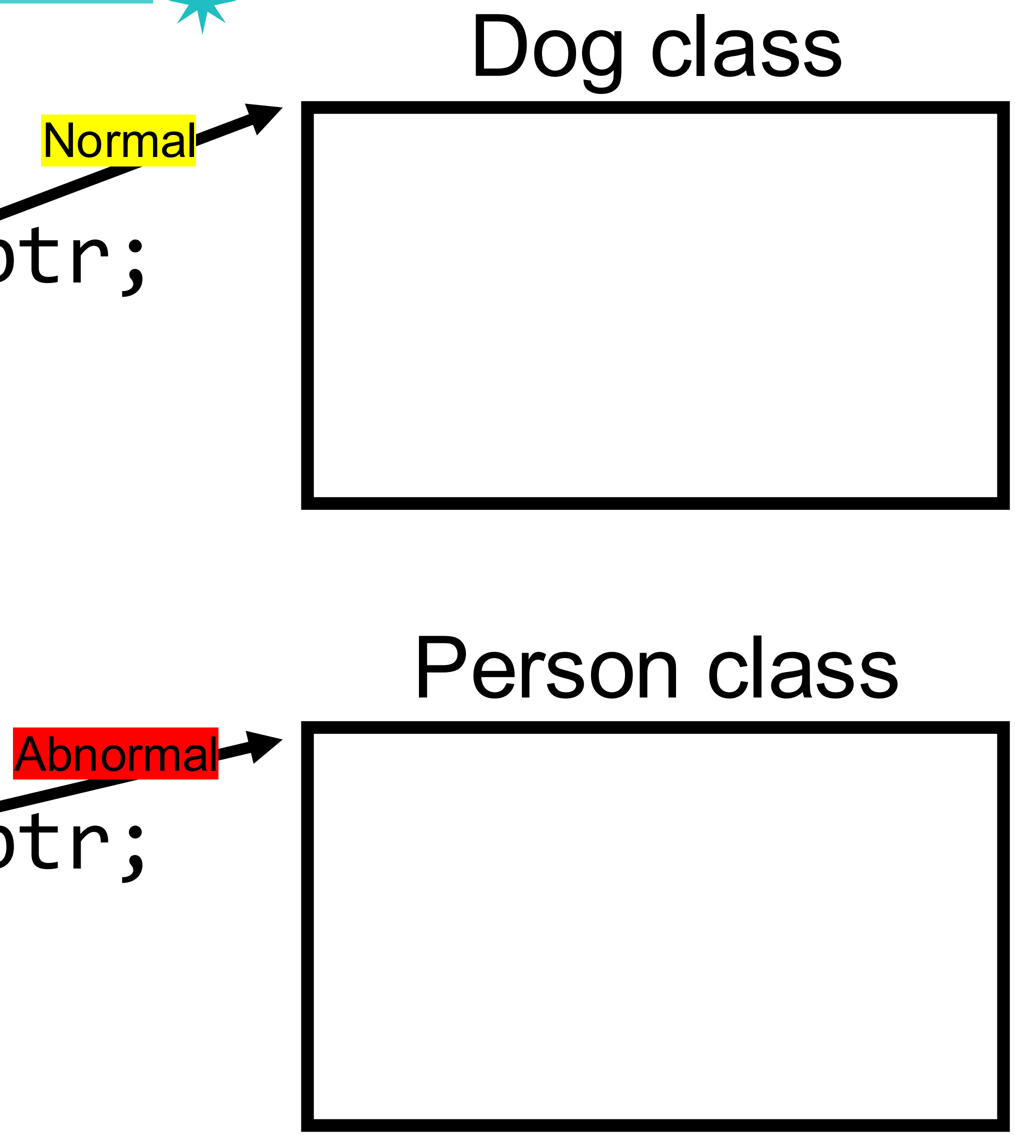


Type Confusion


```
Dog *d = (Dog*) some_ptr;
d->bark();
```

Type Confusion

```
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```



Type Confusion

Dog class

Normal

```
Dog *d = (Dog*) some_ptr;
d->bark();
```

Type Confusion

Person class

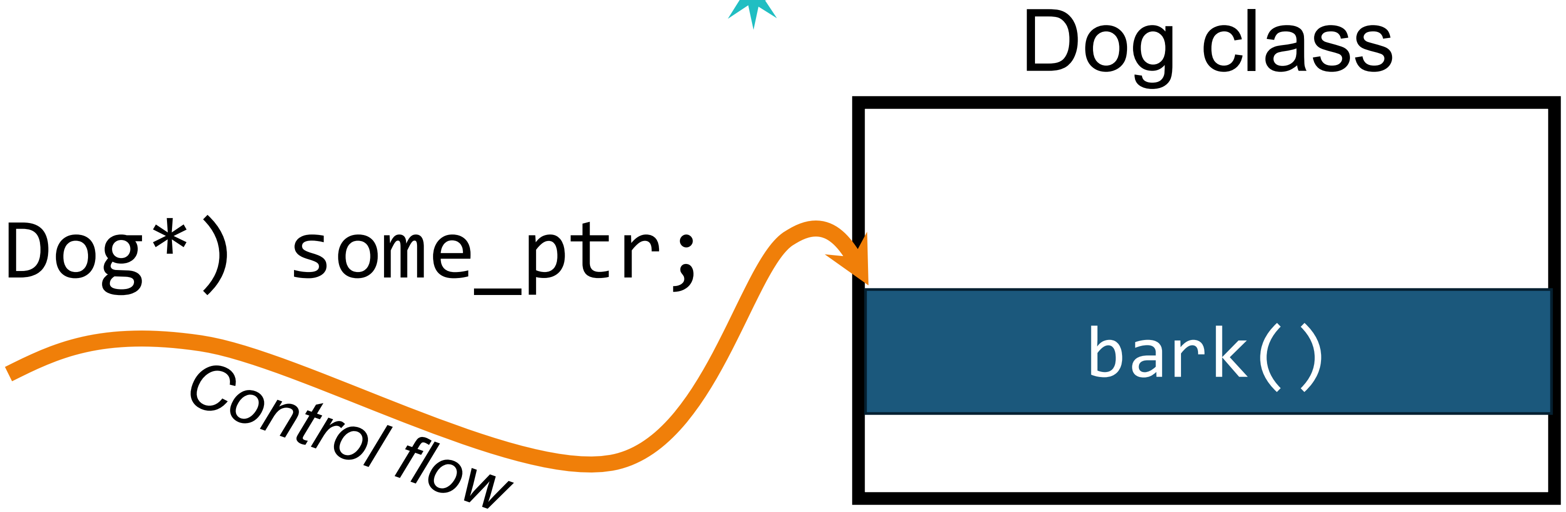
Abnormal

```
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```

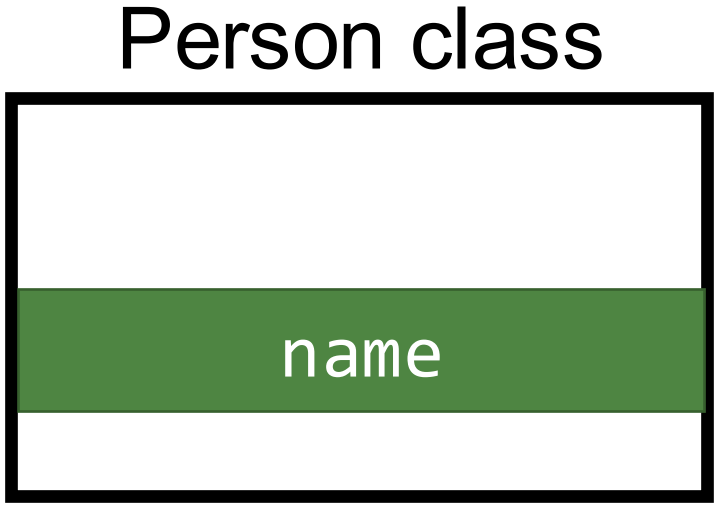
Invoke person's something

Type Confusion Attack (Implication)

```
Dog *d = (Dog*) some_ptr;
d->bark();
```



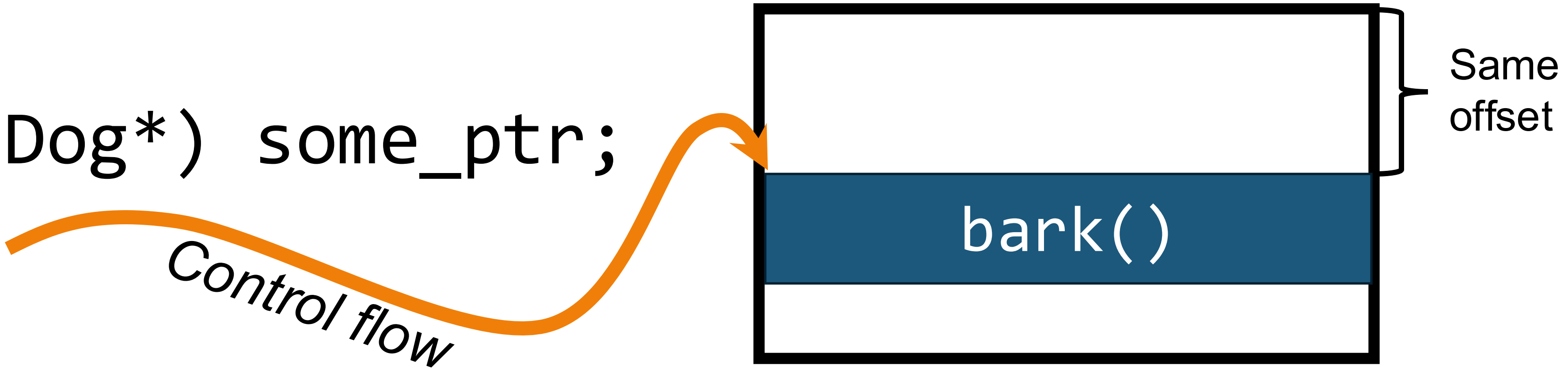

```
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```



Type Confusion Attack (Implication)

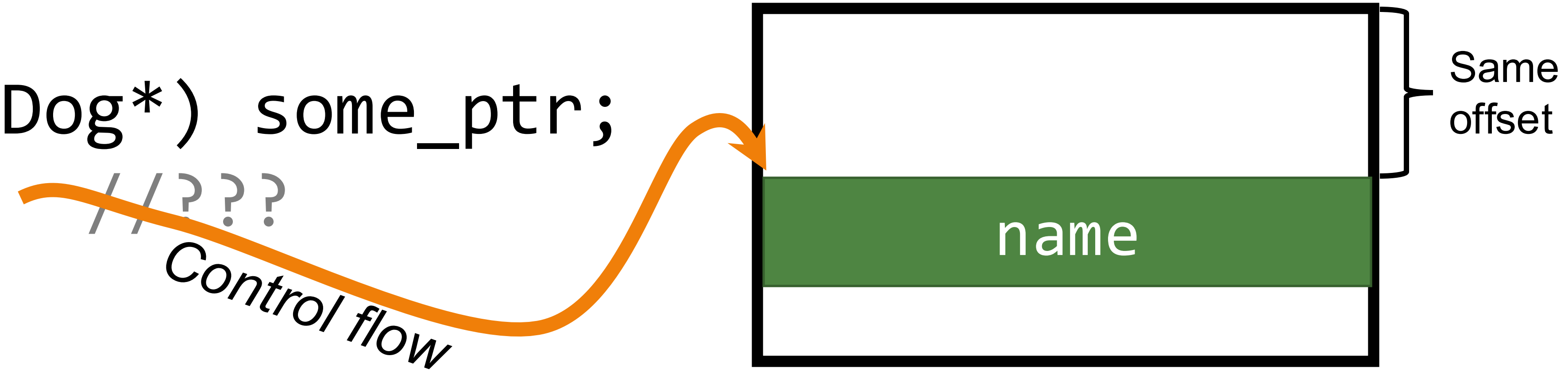
```
Dog *d = (Dog*) some_ptr;
d->bark();
```

Dog class



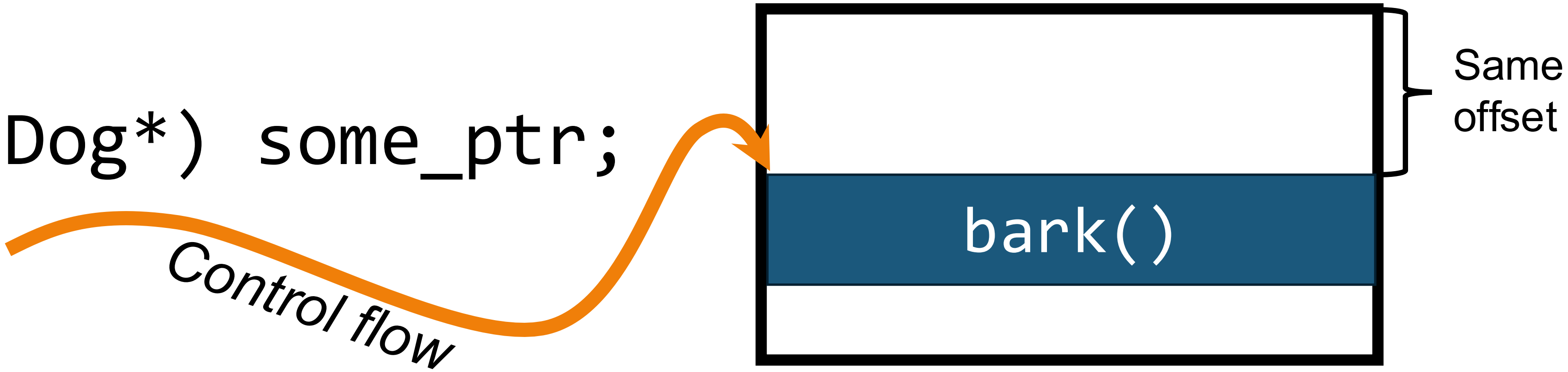

```
Dog *d = (Dog*) some_ptr;
d->bark(); //???
```

Person class



Type Confusion Attack (Implication)

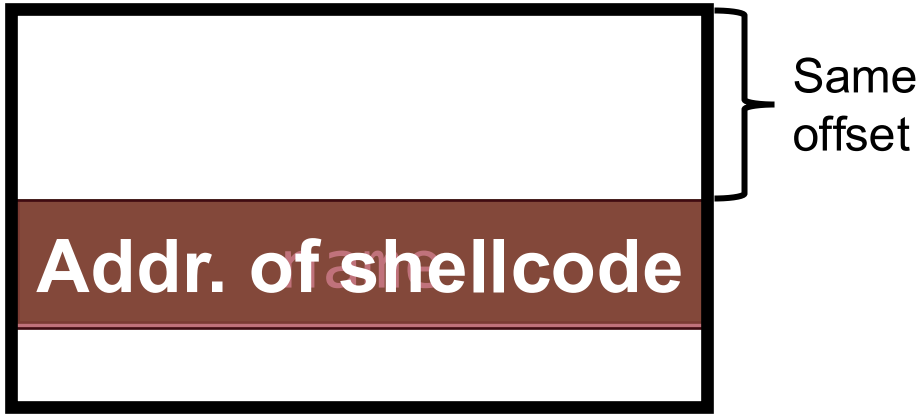
Dog class




```
Dog *d = (Dog*) some_ptr;
d->bark();
```

```
some_ptr->name="[shellcode]"
…
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```

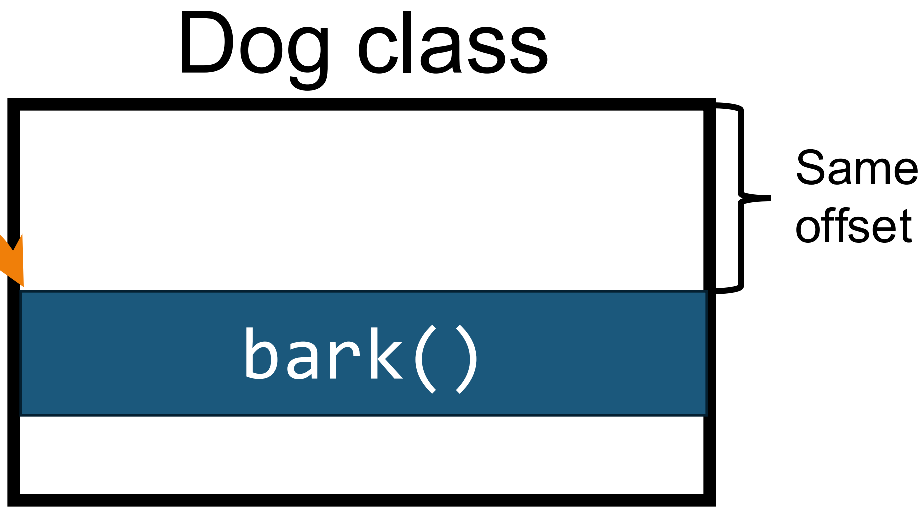
Person class



Type Confusion Attack (Implication)

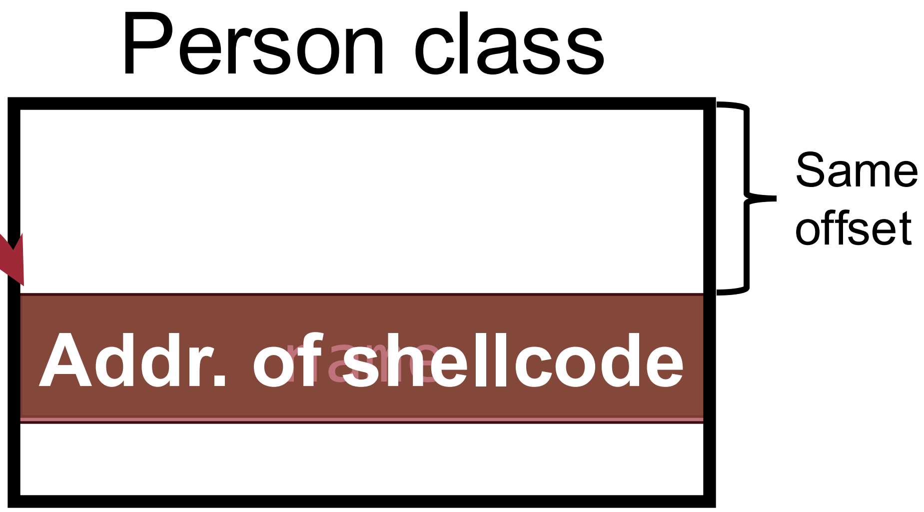

```
Dog *d = (Dog*) some_ptr;
d->bark();
```

Control flow




```
some_ptr->name="[shellcode]"
…
Dog *d = (Dog*) some_ptr;
d->bark(); //???
```

Control flow



Type Confusion Example: Downcasting

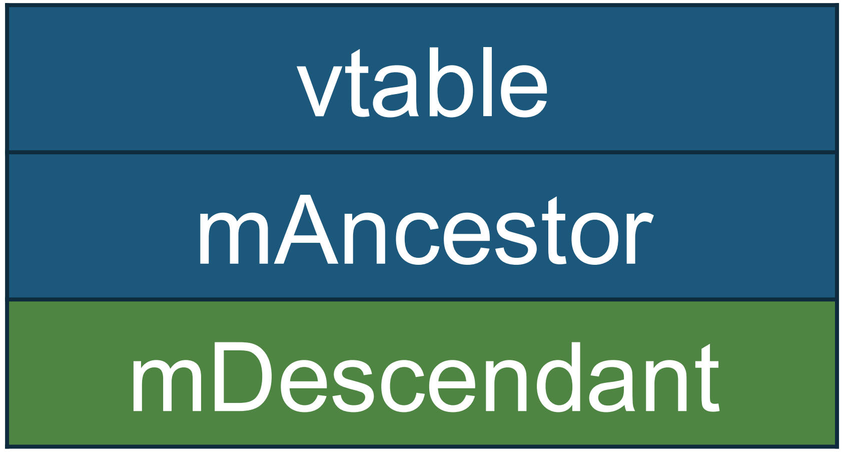
```
class Ancestor {
    public:
        int mAncestor;
    ...
};

class Descendant: public Ancestor {
    public:
        int mDescendant;
    ...
};
```

Inherit
Ancestor class

Type Confusion Example: Downcasting

```
class Ancestor {
    public:
        int mAncestor;
    ...
};
```


```
class Descendant: public Ancestor {
    public:
        int mDescendant;
    ...
};
```



Vulnerable code

```
Ancestor* a = new Ancestor();
Descendant* d = static_cast<Descendant*>(a);
d->mDescendant = 42;
```

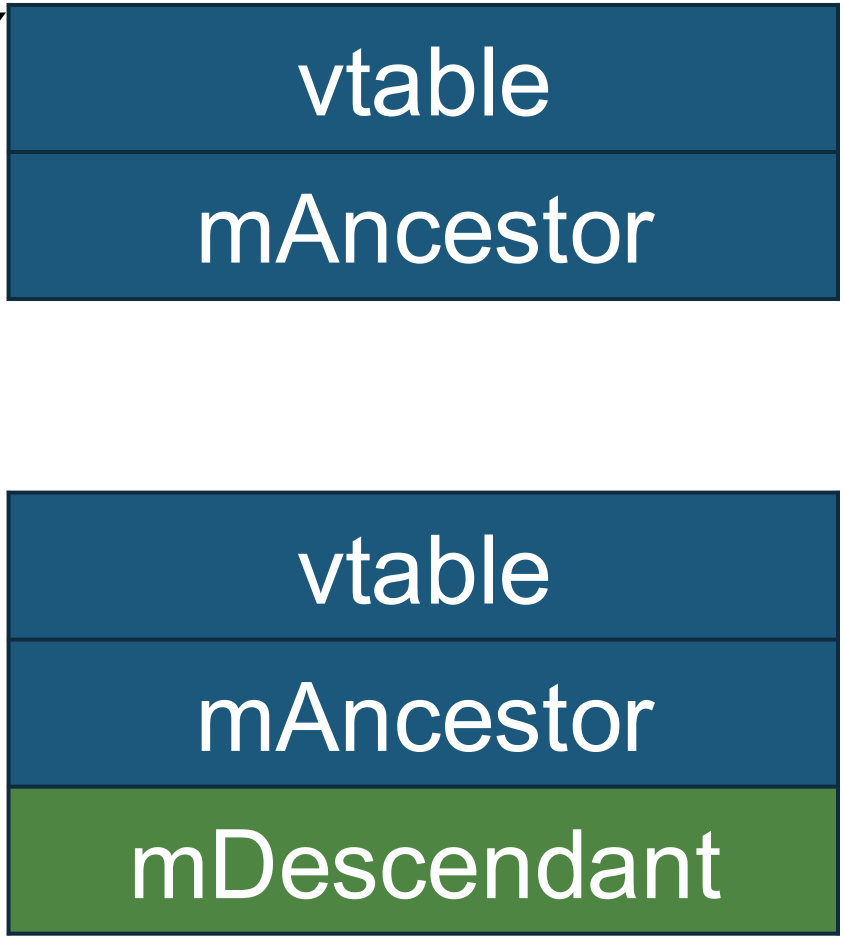
Type Confusion Example: Downcasting

```
class Ancestor {
    public:
        int mAncestor;
    ...
};

class Descendant: public Ancestor {
    public:
        ...scendant;
    ...
};
```

Downcasted pointer

Vulnerable code

```
Ancestor* a = new Ancestor();
Descendant* d = static_cast<Descendant*>(a);
d->mDescendant = 42;
```



Type Confusion Example: Downcasting

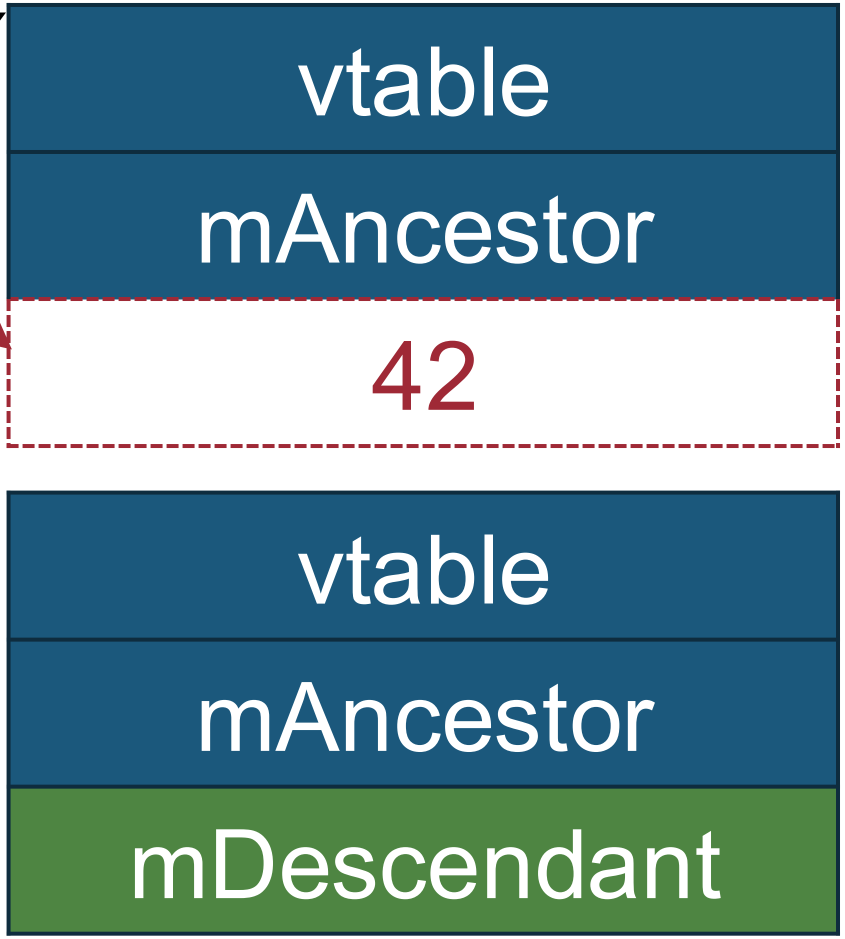
```
class Ancestor {
    public:

};

class Descendant: publi      estor {
    public:
                 scendant;

};
```

Memory corruption:
It can now access a memory region that was not allocated!

Downcasted pointer

Vulnerable code

```
Ancestor* a = new Ancestor();
Descendant* d = static_cast<Descendant*>(a);
d->mDescendant = 42;
```



Question: But, Why Get Confused?

Suppose there is a huge gap between these lines
(e.g., separated in two different libraries)

Vulnerable code

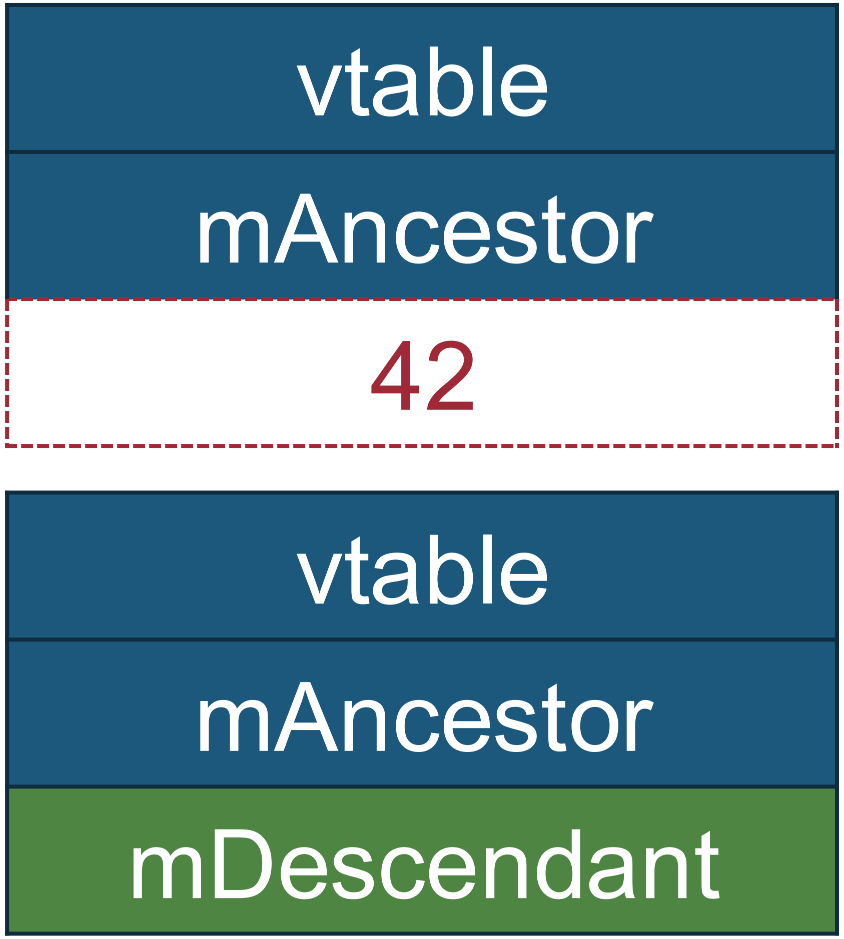
```
Ancestor* a = new Ancestor();
Descendant* d = static_cast<Descendant*>(a);
d->mDescendant = 42;
```

Implication of the Downcasting

What if a user can write an arbitrary value to the confused pointer?

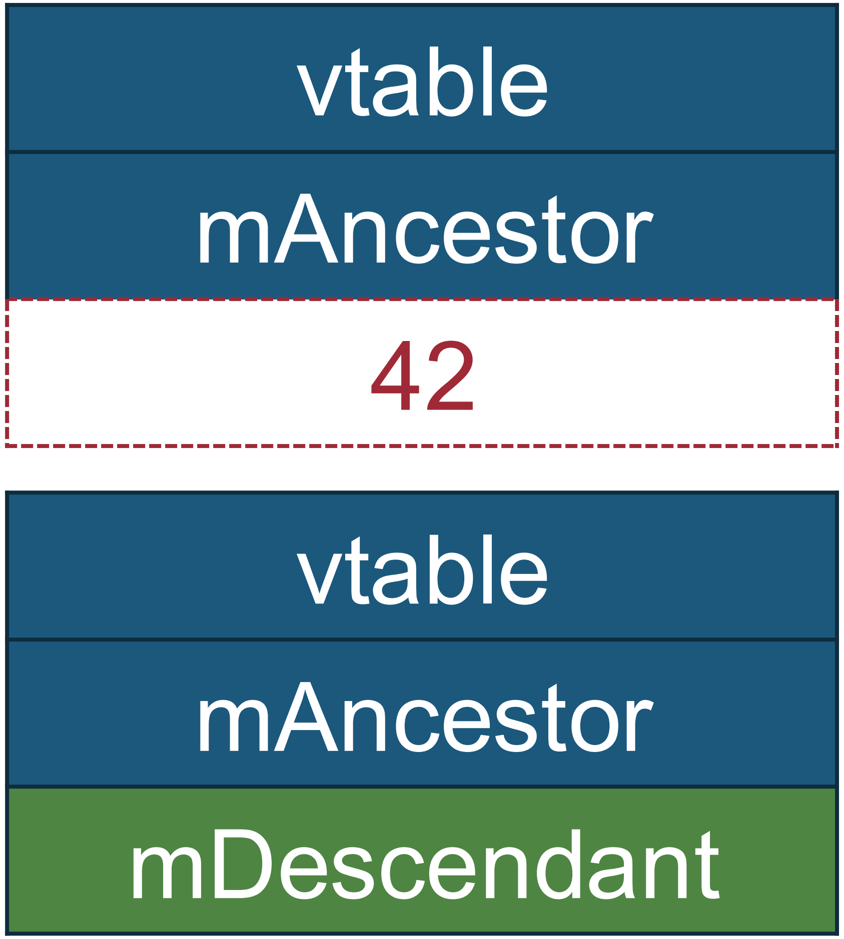
Vulnerable code

```
Ancestor* a = new Ancestor();
Descendant* d = static_cast<Descendant*>(a);
d->mDescendant = 42;
```



Attacker's Perspective

Unlike other attack vectors, we can ***reliably*** corrupt a certain memory field, *i.e.*, we don't need to know the actual address of `mDescendant`



Vulnerable code

```
Ancestor* a = new Ancestor();
Descendant* d = static_cast<Descendant*>(a);
d->mDescendant = 42;
```

Real-world Example (CVE-2013-0912)

- Type Confusion in WebKit (Used in Pwn2Own 2013)

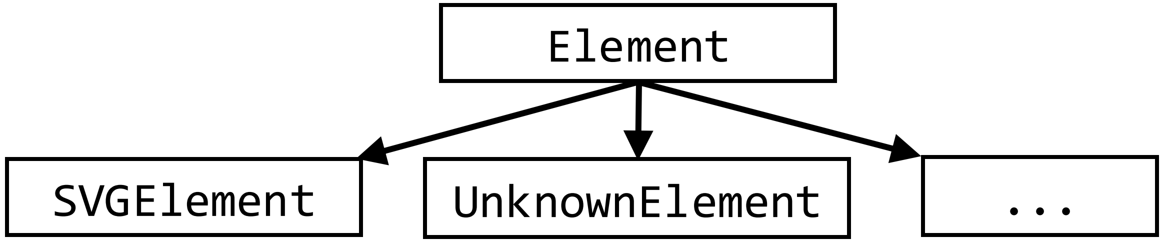
```
SVGElement* SVGViewSpec::viewTarget() {
  if (!m_contextElement)
    return 0;
  return static_cast<SVGElement*>(
    m_contextElement->treeScope()->getElementById(
      m_viewTargetString
    )
  );
}
```

Real-world Example (CVE-2013-0912)

- Type Confusion in WebKit (Used in Pwn2Own 2013)

```
SVGElement* SVGViewSpec::viewTarget() {
  if (!m_contextElement)
    return 0;
  return static_cast<SVGElement*>(
    m_contextElement->treeScope()->getElementById(
      m_viewTargetString
    )
  );
}
```

Developer thought this must always be SVGElement type
(but it turned out to be not!)

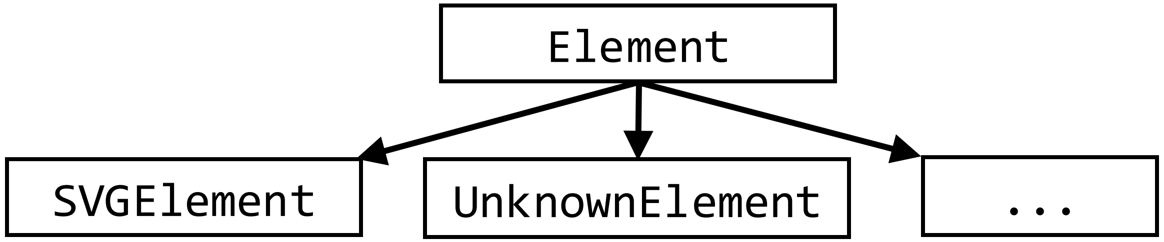


Real-world Example (CVE-2013-0912)

- Type Confusion in WebKit (Used in Pwn2Own 2013)

```
SVGElement* SVGViewSpec::viewTarget() {
  if (!m_contextElement)
    return 0;
  return static_cast<SVGElement*>(
    m_contextElement->treeScope()->getElementById(
      m_viewTargetString
    )
  );
}
```

Developer thought this must always be SVGElement type (but it turned out to be not!)



PoC.html

```
<svg xmlns="…"
  <foreignobject>
    <body xmlns="…">
        <feColorMatrix viewTarget feColorMatrix>
    </body>
  </foreignobject>
</svg>
```

Real-world Example (CVE-2013-0912)

- Type Confusion in WebKit (Used in Pwn2Own 2013)

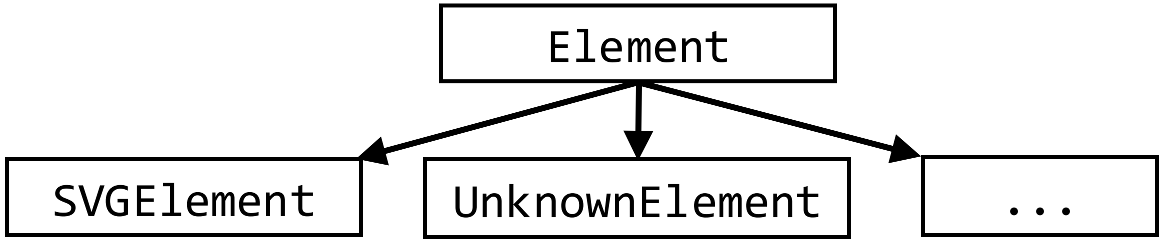
```
SVGElement* SVGViewSpec::viewTarget() {
  if (!m_contextElement)
    return 0;
  return static_cast<SVGElement*>(
    m_contextElement->treeScope()->getElementById(
      m_viewTargetString
    )
  );
}
```

Developer thought this must always be SVGElement type (but it turned out to be not!)

Return UnknownElement

PoC.html

```
<svg xmlns="…"
  <foreignobject>
    <body xmlns="…">
        <feColorMatrix viewTarget feColorMatrix>
    </body>
  </foreignobject>
</svg>
```



Patch: Use `dynamic_cast`

Limitations:

- Slow
- Compiler options such as `--fno-rtti` can disable it!

Use After Free
(A popular source of type confusion)

Use After Free

- If after freeing a memory location, a program does not clear the pointer to that memory, an attacker can use it to hack the program

Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

Class information

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

Allocate a memory block on the heap

Class information

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

Class Foo

f

Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

Class information

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

Use After Free Example


```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

Class information

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

Return the block to the free list

Class information

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

Class information

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

Class Foo

Foo.x = 42

ptr

Often called *“Dangling Pointer”*

Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

Find an appropriate block from the list of free blocks

Class information

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

Class information

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

Class Bar

Bar.y=“hello world”

b

ptr

Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

Class information

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

Class Bar

Bar.y=“hello world”

b

ptr

Print the address of the Bar.y

Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = "hello world";
cout << ptr->x << endl;
```

Class information

```
class Foo {
  public:
    int x;
}
ar* y;
};
```

We *used* this pointer ***after free***

b

ptr

Class Bar

Bar.y="hello world"

Print the address of the Bar.y

Use After Free can Trigger Type Confusion

- A dangling pointer’s type and the corresponding reallocated data’s type can be different => Trigger type confusion!

Example: OpenSSL UAF Bug


```
...
dtls1_hm_fragment_free(frag);
pitem_free(item);
if (al==0) {
    *ok = 1;
    return frag->msg_header.frag_len;
}
```

Example: OpenSSL UAF Bug

```
...
dtls1_hm_fragment_free(frag);
pitem_free(item);
if (al==0) {
    *ok = 1;
    return frag->msg_header.frag_len;
}
```

frag is freed

Example: OpenSSL UAF Bug

```
...
dtls1_hm_fragment_free(frag);
pitem_free(item);
if (al==0) {
    *ok = 1;
    return frag->msg_header.frag_len;
}
```

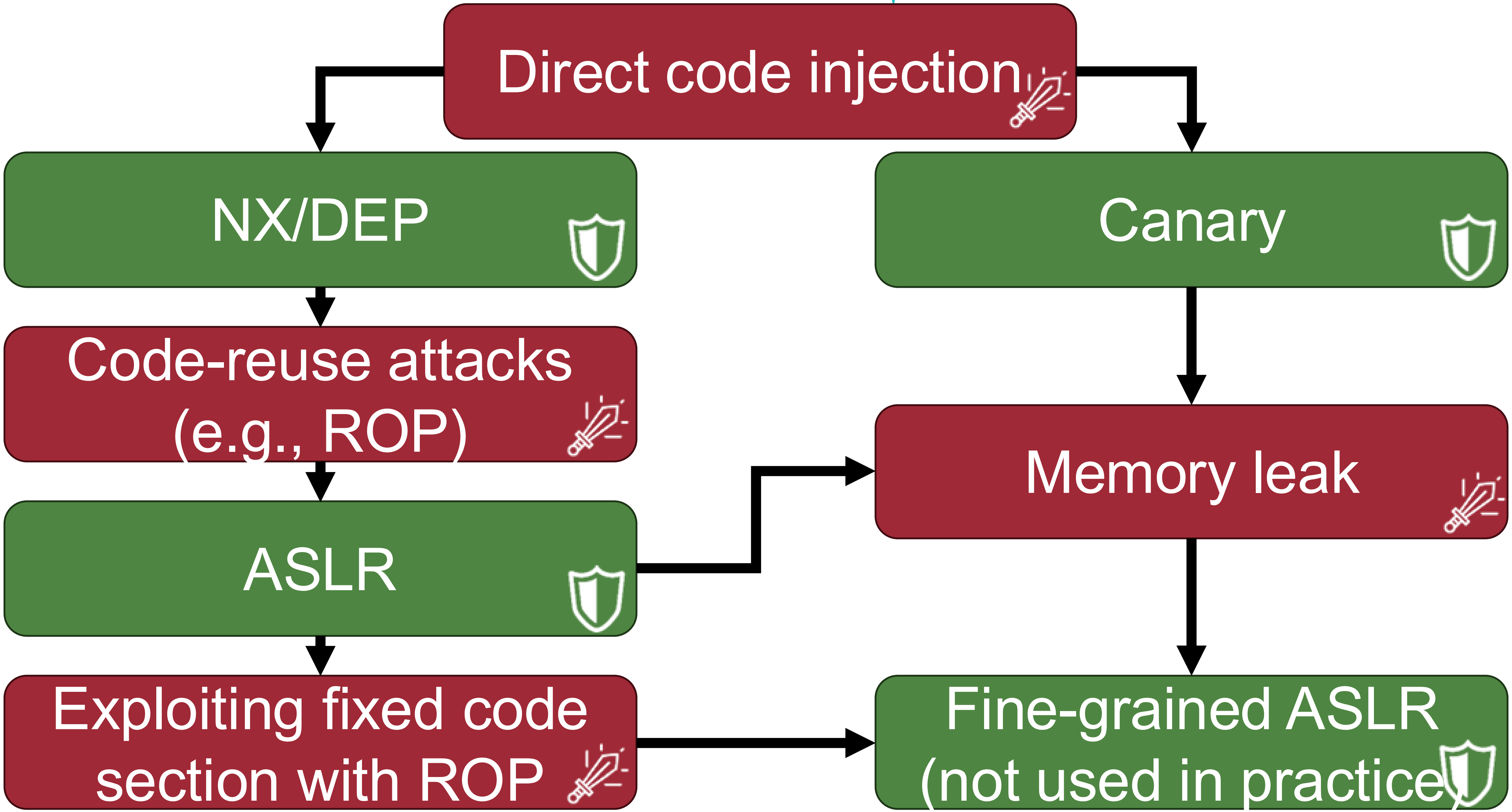
frag is freed

Read after the free

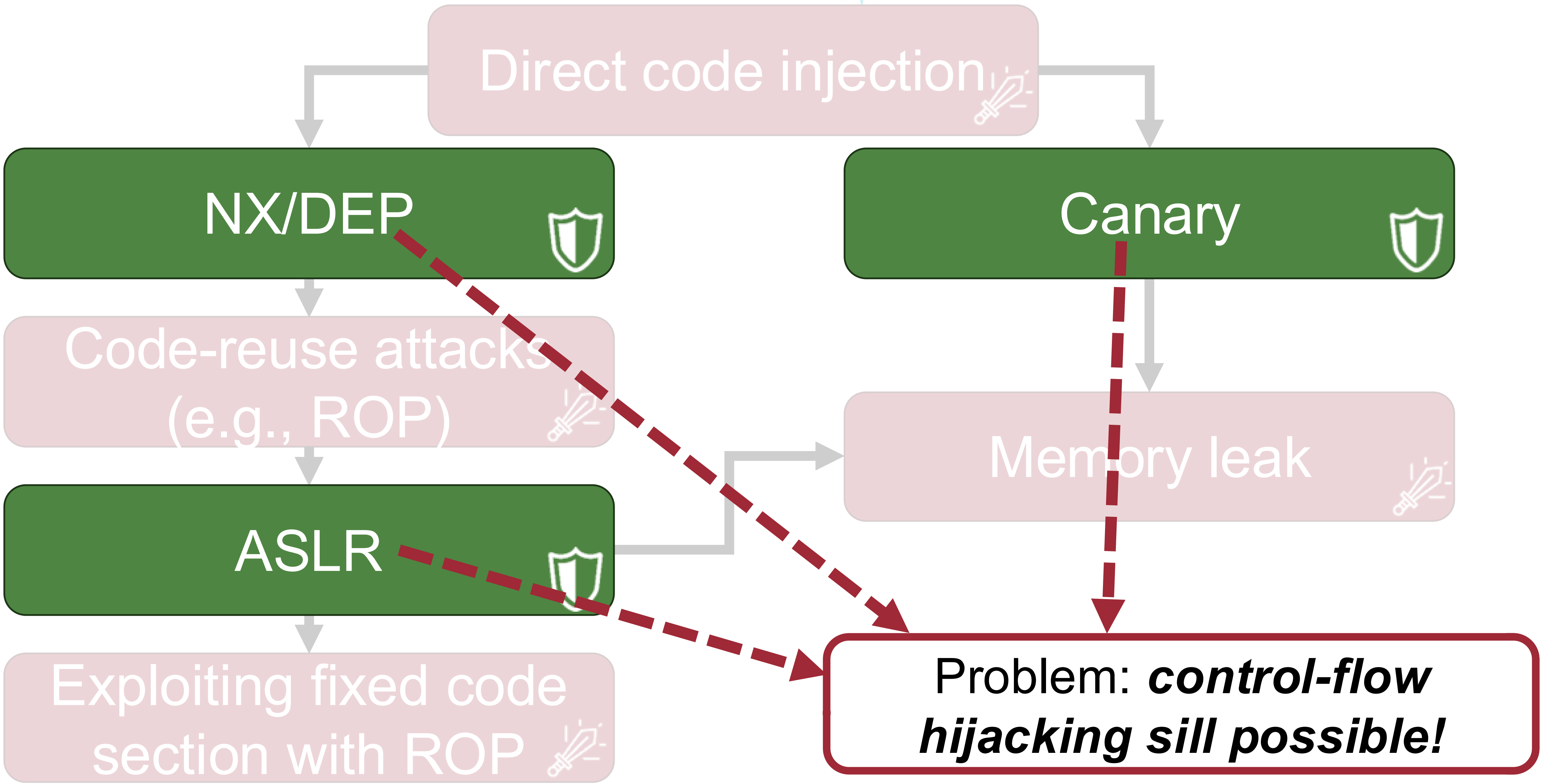
Research: Type Confusion Detection

- Static Detection of C++ vtable Escape Vulnerabilities in Binary Code, ***NDSS 2012***
- Type Casting Verification: Stopping an Emerging Attack Vector, ***USENIX Security 2015***

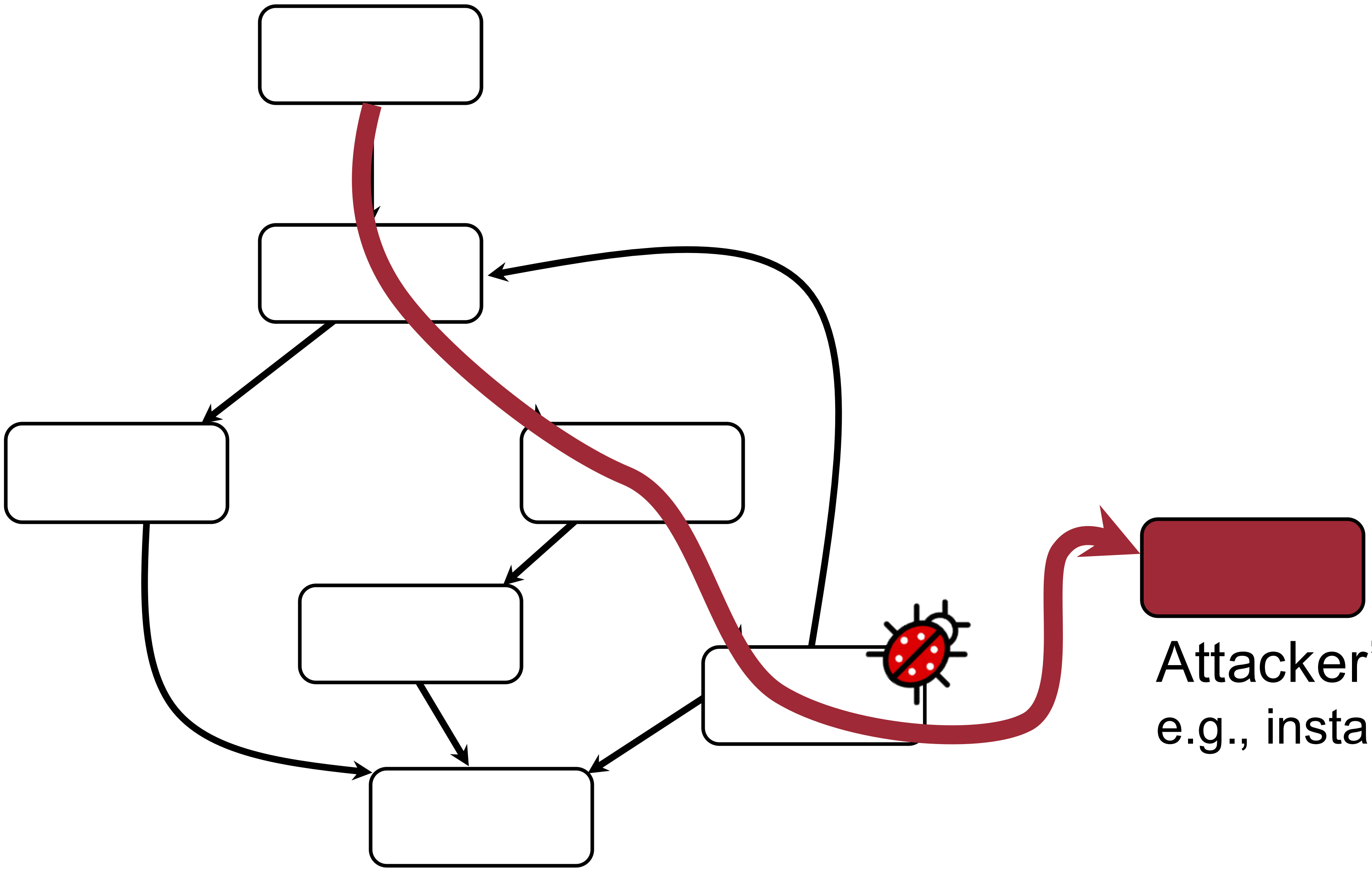
Attack / Defense So Far



Problems of the Current Defenses



Control Flow Hijack Exploit



Attacker’s own code/logic
e.g., install malicious software

Can we enforce control-flow integrity?

Control Flow Integrity (CFI)

CFI Policy

The CFI security policy dictates that software execution must follow a path of a Control-Flow Graph (CFG) determined ***ahead of time***.

Control-flow Integrity, ***CCS 2005***

Control-Flow Integrity

Principles, Implementations, and Applications

Martín Abadi
Computer Science Dept.
University of California
Santa Cruz

Mihai Budiu   Úlfar Erlingsson
Microsoft Research
Silicon Valley

Jay Ligatti
Dept. of Computer Science
Princeton University

ABSTRACT

Current software attacks often build on exploits that subvert machine-code execution. The enforcement of a basic safety property, Control-Flow Integrity (CFI), can prevent such attacks from arbitrarily controlling program behavior. CFI enforcement is simple, and its guarantees can be established formally, even with respect to powerful adversaries. Moreover, CFI enforcement is practical: it is compatible with existing software and can be done efficiently using software rewriting in commodity systems. Finally, CFI provides a useful foundation for enforcing further security policies, as we demonstrate with efficient software implementations of a protected shadow call stack and of access control for memory regions.

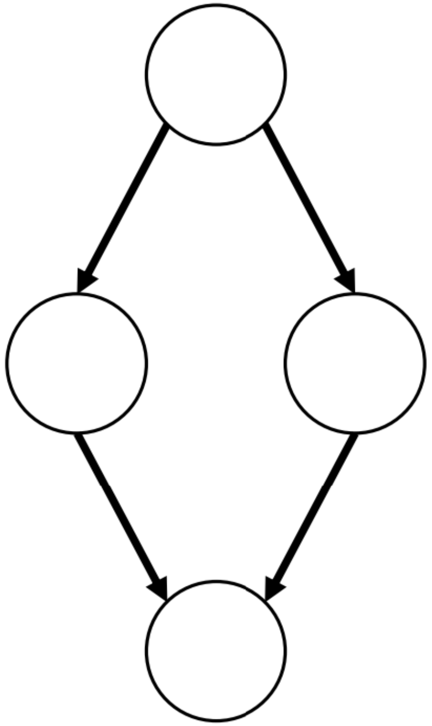
bined effects of these attacks make them one of the most pressing challenges in computer security.

In recent years, many ingenious vulnerability mitigations have been proposed for defending against these attacks; these include stack canaries [14], runtime elimination of buffer overflows [46], randomization and artificial heterogeneity [41, 62], and tainting of suspect data [55]. Some of these mitigations are widely used, while others may be impractical, for example because they rely on hardware modifications or impose a high performance penalty. In any case, their security benefits are open to debate: mitigations are usually of limited scope, and attackers have found ways to circumvent each deployed mitigation mechanism [42, 49, 61].

The limitations of these mechanisms stem, in part, from the lack

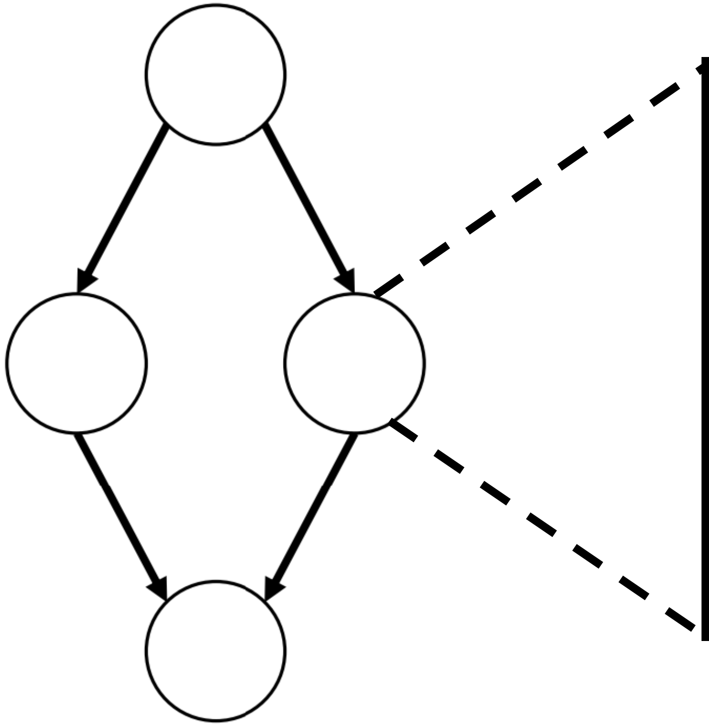
Background: Control Flow Graph (CFG)

- A CFG is a graph that represents all paths that might be traversed through a program execution



Background: Control Flow Graph (CFG)

- A CFG is a graph that represents all paths that might be traversed through a program execution
- Each node in a CFG represents a basic block



Basic Block:

A sequence of statements that is always entered at the beginning and exited at the end*

Background: Basic Block

```
 0:   55                      push    ebp
 1:   89 e5                   mov     ebp,esp
 3:   83 ec 10                sub     esp,0x10
 6:   c7 45 f8 00 00 00 00    mov     DWORD PTR [ebp-0x8],0x0
 d:   c7 45 fc 0a 00 00 00    mov     DWORD PTR [ebp-0x4],0xa
14:   eb 08                   jmp     1e <v+0x1e>
16:   83 45 f8 01             add     DWORD PTR [ebp-0x8],0x1
1a:   83 6d fc 01             sub     DWORD PTR [ebp-0x4],0x1
1e:   83 7d fc 00             cmp DWORD PTR [ebp-0x4],0x0
22:   7f f2                   jg      16 <v+0x16>
24:   8b 45 f8                mov eax,DWORD PTR [ebp-0x8]
27:   c9                      leave
28:   c3                      ret
```

Background: Basic Block

```
 0:   55                          push   ebp
 1:   89 e5                       mov    ebp,esp
 3:   83 ec 10                    sub    esp,0x10
 6:   c7 45 f8 00 00 00 00        mov    DWORD PTR [ebp-0x8],0x0
 d:   c7 45 fc 0a 00 00 00        mov    DWORD PTR [ebp-0x4],0xa
14:   eb 08                       jmp    1e <v+0x1e>
16:   83 45 f8 01                 add    DWORD PTR [ebp-0x8],0x1
1a:   83 6d fc 01                 sub    DWORD PTR [ebp-0x4],0x1
1e:   83 7d fc 00                 cmp DWORD PTR [ebp-0x4],0x0
22:   7f f2                       jg     16 <v+0x16>
24:   8b 45 f8                    mov eax,DWORD PTR [ebp-0x8]
27:   c9                          leave
28:   c3                          ret
```

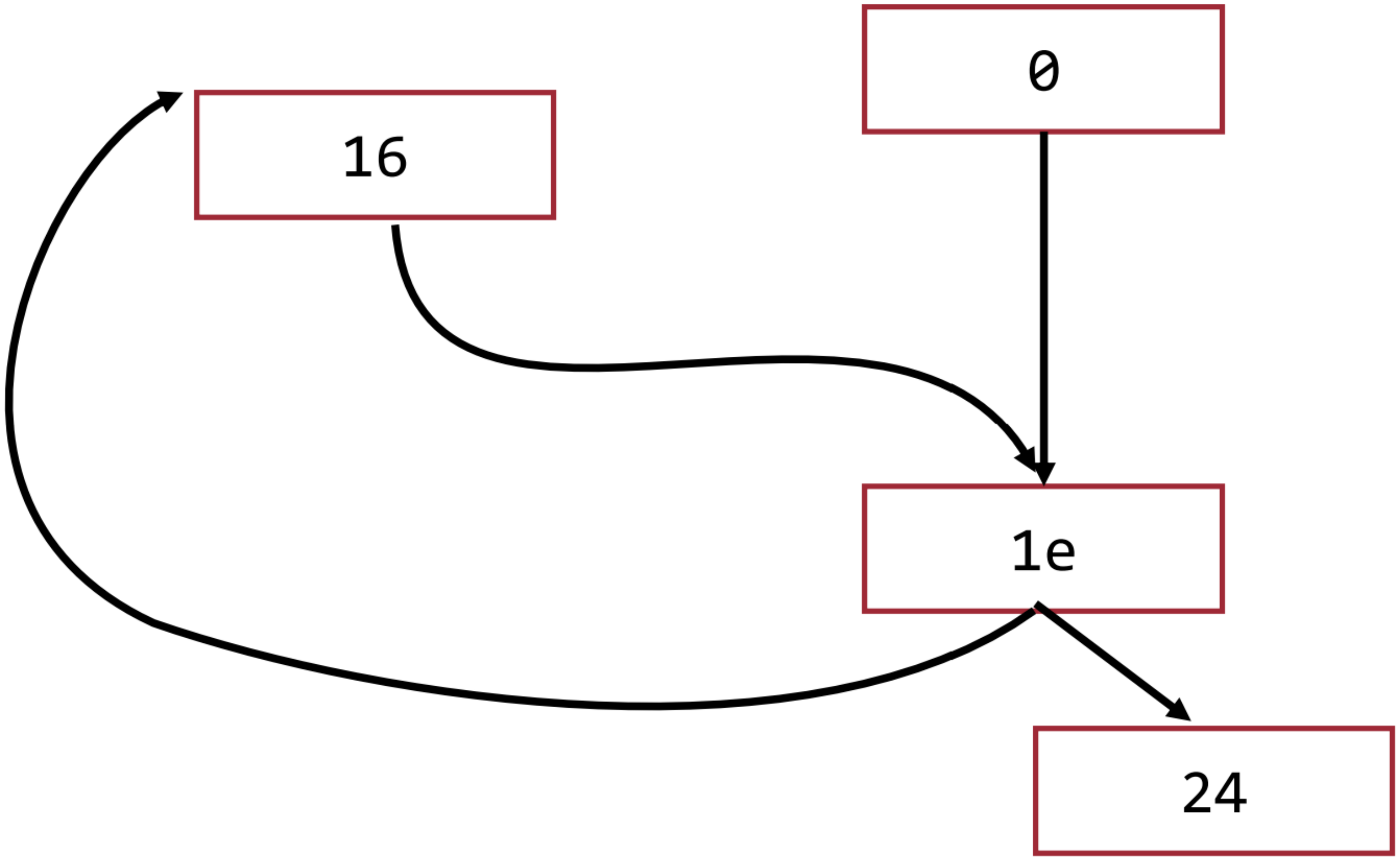
Background: Basic Block

```
 0:   55                        push   ebp
 1:   89 e5                     mov    ebp,esp
 3:   83 ec 10                  sub    esp,0x10
 6:   c7 45 f8 00 00 00 00      mov    DWORD PTR [ebp-0x8],0x0
 d:   c7 45 fc 0a 00 00 00      mov    DWORD PTR [ebp-0x4],0xa
14:   eb 08                     jmp    1e <v+0x1e>
16:   83 45 f8 01               add    DWORD PTR [ebp-0x8],0x1
1a:   83 6d fc 01               sub    DWORD PTR [ebp-0x4],0x1
1e:   83 7d fc 00               cmp DWORD PTR [ebp-0x4],0x0
22:   7f f2                     jg     16 <v+0x16>
24:   8b 45 f8                  mov eax,DWORD PTR [ebp-0x8]
27:   c9                        leave
28:   c3                        ret
```

Background: Basic Block

```
 0:   55                          push   ebp
 1:   89 e5                       mov    ebp,esp
 3:   83 ec 10                    sub    esp,0x10
 6:   c7 45 f8 00 00 00 00        mov    DWORD PTR [ebp-0x8],0x0
 d:   c7 45 fc 0a 00 00 00        mov    DWORD PTR [ebp-0x4],0xa
14:   eb 08                       jmp    1e <v+0x1e>

16:   83 45 f8 01                 add    DWORD PTR [ebp-0x8],0x1
1a:   83 6d fc 01                 sub    DWORD PTR [ebp-0x4],0x1

1e:   83 7d fc 00                 cmp DWORD PTR [ebp-0x4],0x0
22:   7f f2                       jg     16 <v+0x16>

24:   8b 45 f8                    mov eax,DWORD PTR [ebp-0x8]
27:   c9                          leave
28:   c3                          ret
```

Control Flow Graph of the Example



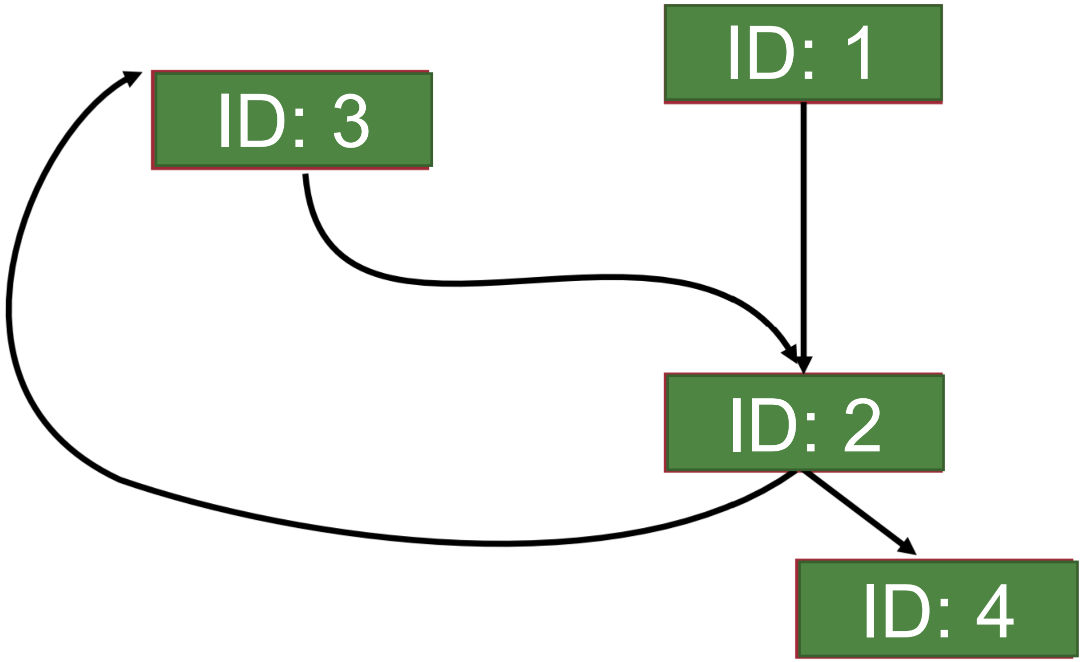
CFI Intuition: Any execution should follow control paths of this CFG!

CFI Assumptions

- Attackers cannot execute data (DEP is enabled)
- Programs cannot change themselves (no self-modifying code)

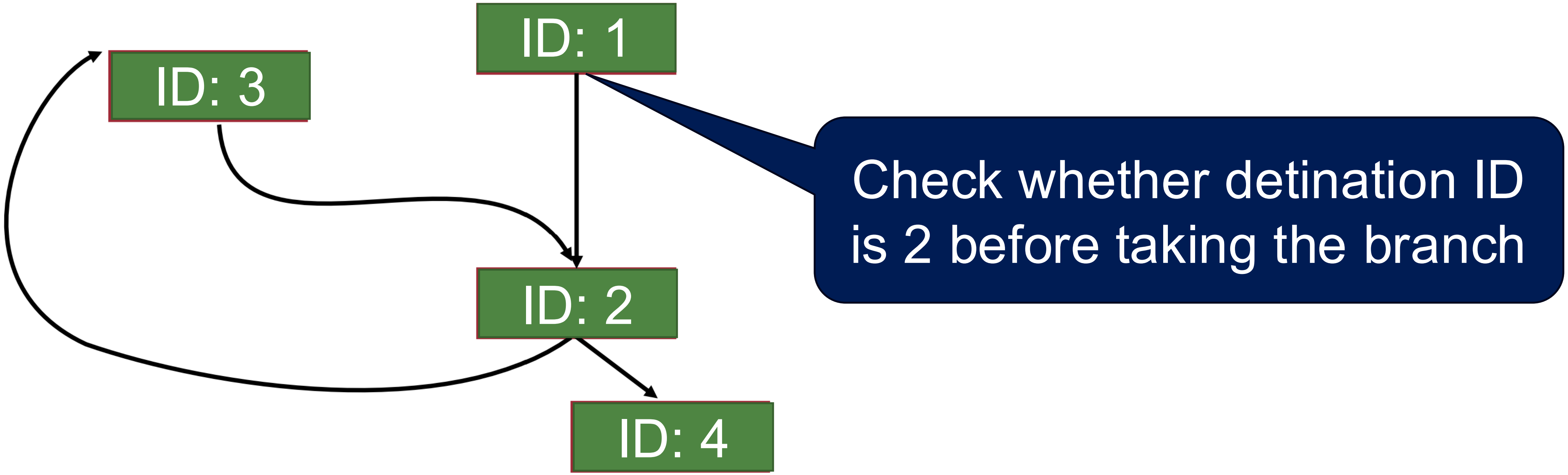
How to Enforce CFI?

- Give unique IDs at destinations



How to Enforce CFI?

- Give unique IDs at destinations
- For all branch instructions, check destination IDs ***before taking the branch***



How to Instrument?

Source

```
Opcode bytes            Instructions
FF E1                   jmp   ecx                   ; computed jump
```

Destination

```
Opcode bytes            Instructions
8B 44 24 04             mov   eax, [esp+4]    ; dst
...
```

can be instrumented as (a):

```
81 39 78 56 34 12       cmp   [ecx], 12345678h      ; comp ID & dst
75 13                   jne   error_label           ; if != fail
8D 49 04                lea   ecx, [ecx+4]          ; skip ID at dst
FF E1                   jmp   ecx                   ; jump to dst
```

```
78 56 34 12             ; data 12345678h      ; ID
8B 44 24 04             mov   eax, [esp+4]    ; dst
...
```

CFI Challenge

What if a single branch instruction can jump to multiple addresses? (e.g., `call eax`)

Example

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

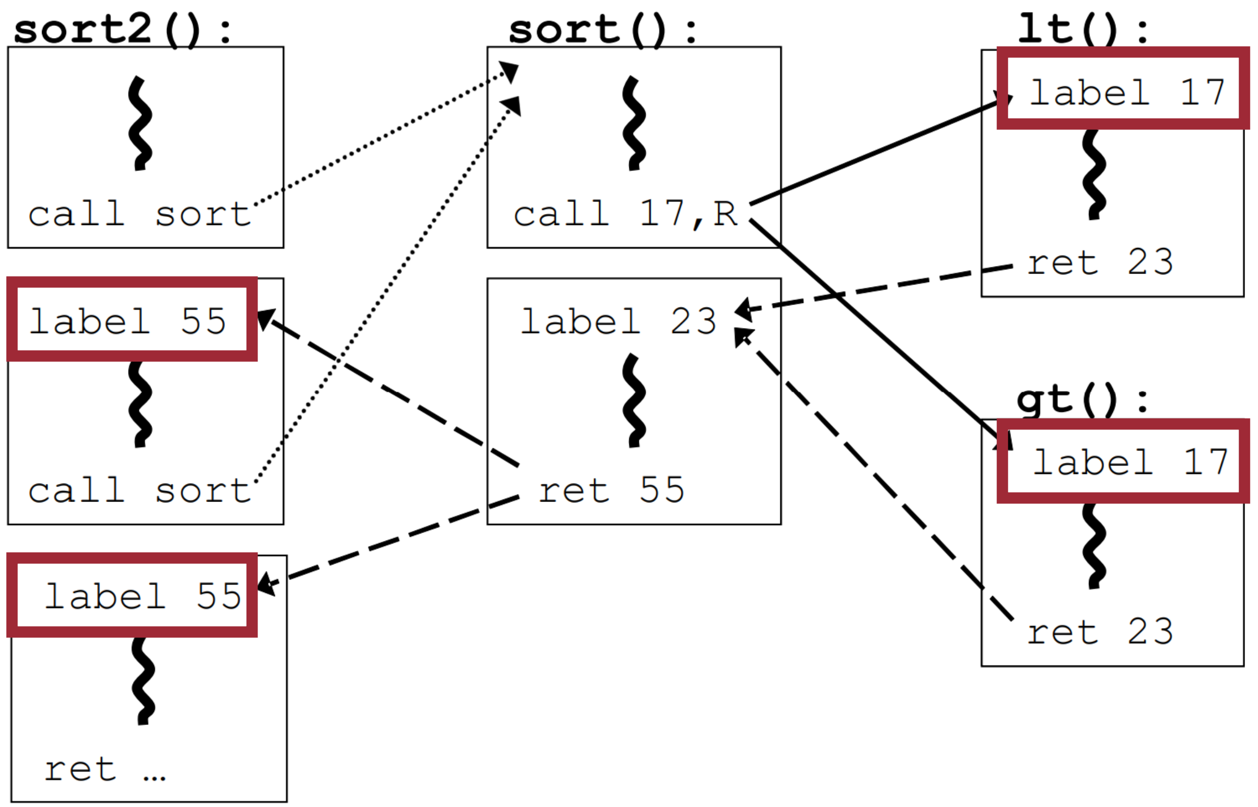
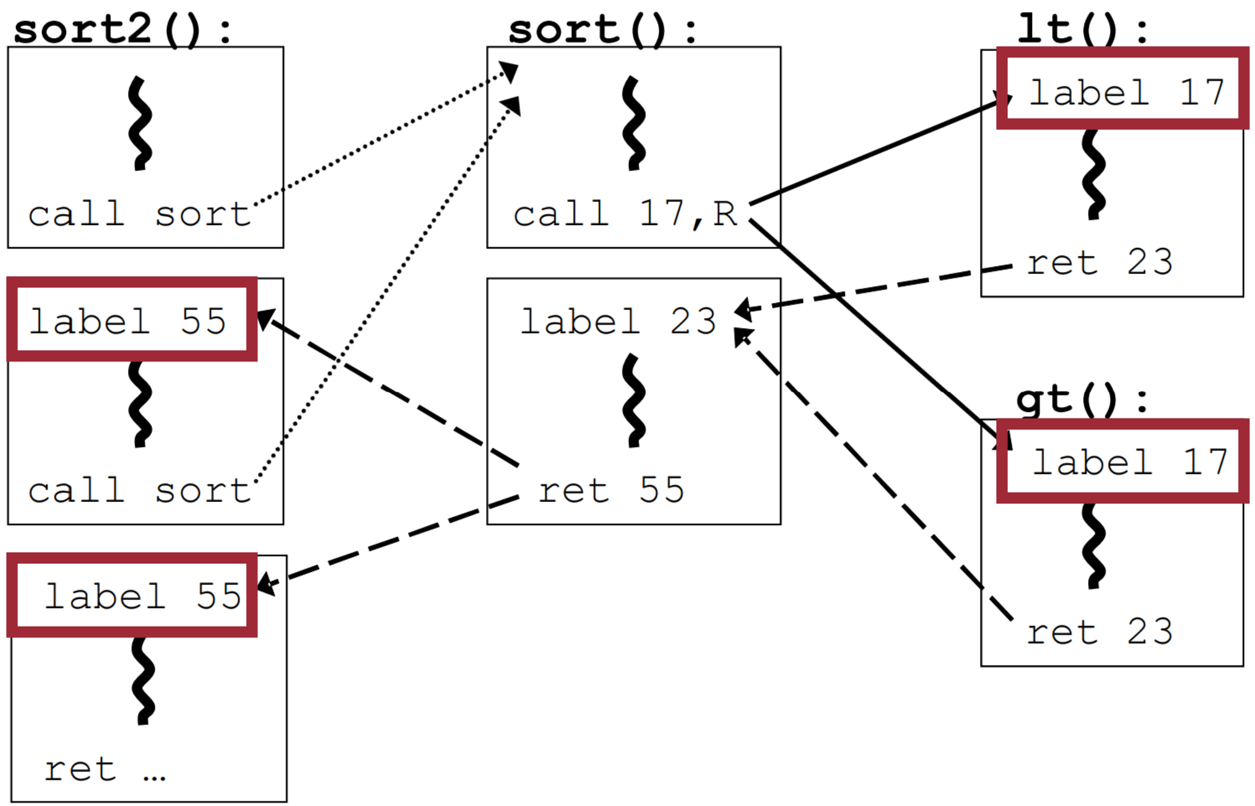


Image from control flow integrity, CCS 2005

Example

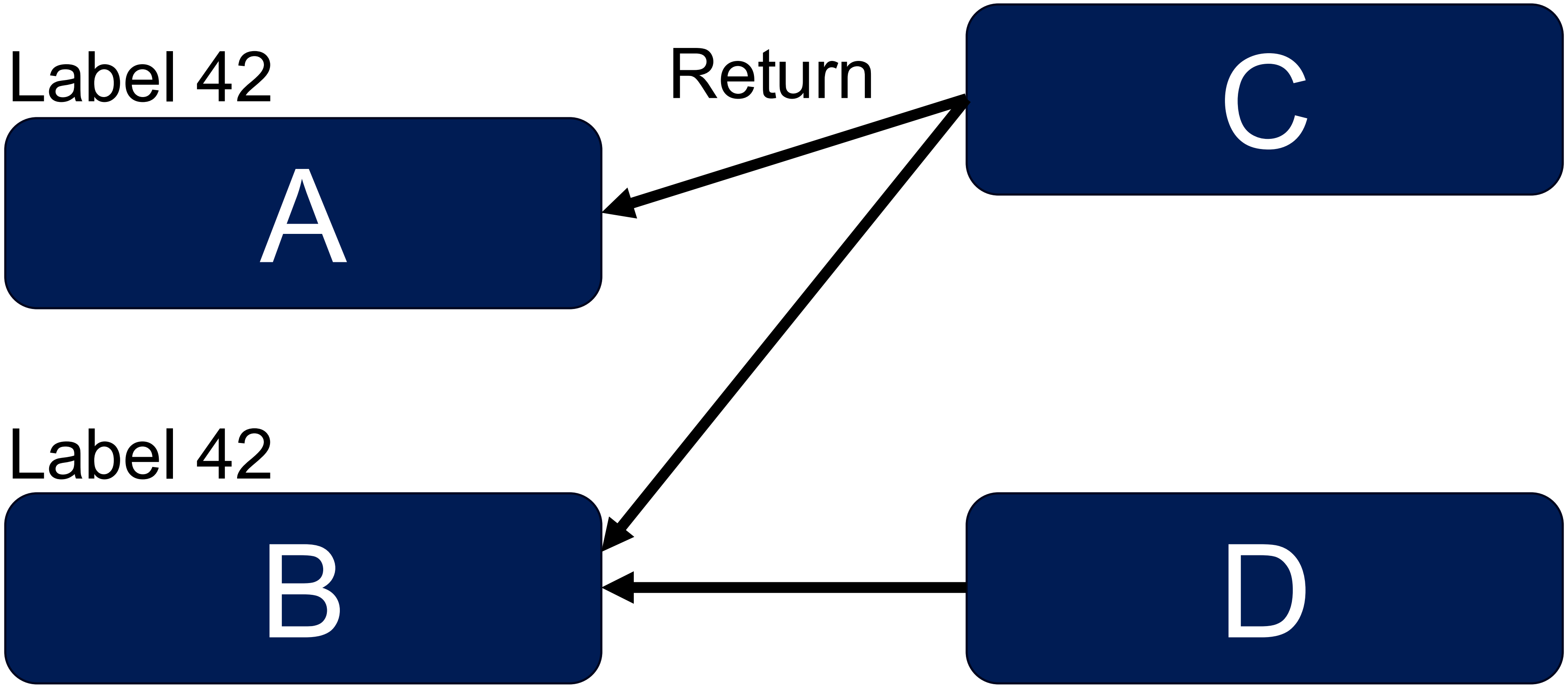

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



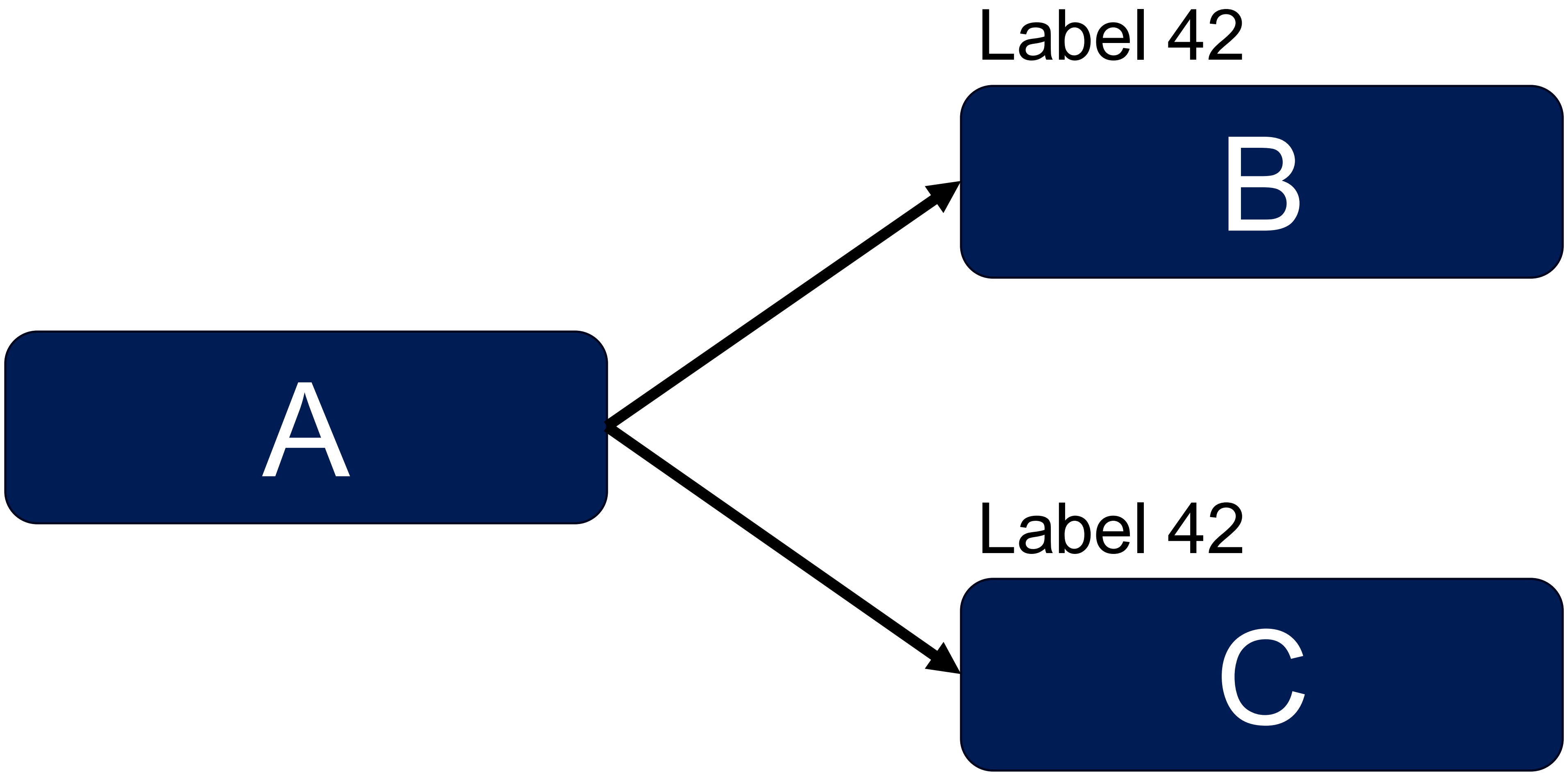
Can you spot labeling problems?

Image from control flow integrity, CCS 2005

Problem #1: What if D returns to A?



Problem #2: Context Insensitive!



Potential Solutions

- Multiple tags
  - Q. What's the problem?

- Shadow call stack

Shadow Call Stack

- **In function prologues**, store the return address in another area of memory

- **In function epilogues**, check if we are returning to the proper address

A Binary Rewriting Defense against Stack based Buffer Overflow Attacks, ***USENIX ATC 2003***

CFI with Shadow Call Stack


```
call  eax            ; call func ptr        ret                ; return
```

with a CFI-based implementation of a protected shadow call stack using hardware segments, can become:

```
      add   gs:[0h], 4h      ; inc stack by 4       mov  ecx, gs:[0h]    ; get top offset
      mov   ecx, gs:[0h]     ; get top offset       mov  ecx, gs:[ecx]   ; pop return dst
      mov   gs:[ecx], LRET   ; push ret dst         sub  gs:[0h], 4h     ; dec stack by 4
      cmp   [eax+4], ID      ; comp fptr w/ID       add  esp, 4h         ; skip extra ret
      jne   error_label      ; if != fail           jmp  ecx             ; jump return dst
      call  eax              ; call func ptr
LRET: ...
```

CFI with Shadow Call Stack


```
call  eax               ; call func ptr
```

```
ret                     ; return
```

with a CFI-based implementation of a protected shadow call stack using hardware segments, can become:

```
add   gs:[0h], 4h       ; inc stack by 4
mov   ecx, gs:[0h]      ; get top offset
mov   gs:[ecx], LRET    ; push ret dst
cmp   [eax+4], ID       ; comp fptr w/ID
jne   error_label       ; if != fail
call  eax               ; call func ptr
LRET: ...
```

```
mov   ecx, gs:[0h]      ; get top offset
mov   ecx, gs:[ecx]     ; pop return dst
sub   gs:[0h], 4h       ; dec stack by 4
add   esp, 4h           ; skip extra ret
jmp   ecx               ; jump return dst
```

Push return address to the shadow call stack

Runtime Overhead

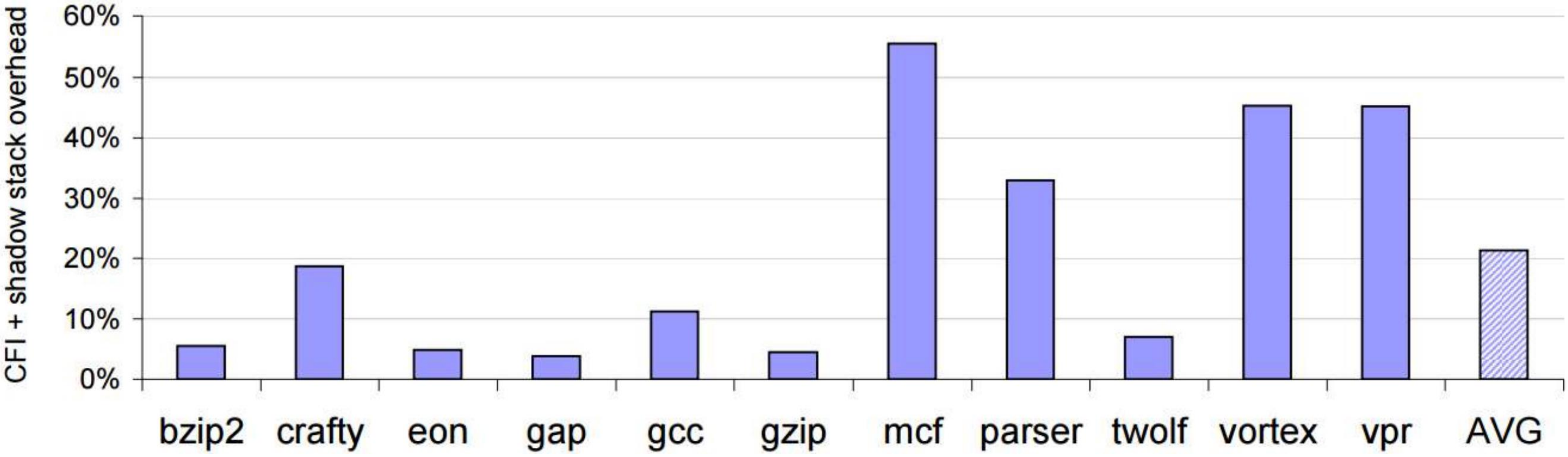


Image from control flow integrity, CCS 2005

CFI Practical Implication?

- CFI on binary code is difficult
  - Subtlety of Vulcan
- CFI is slow

CFI on Binary: Bypassing CFI

- Dynamically generated code
  - Self modifying code (e.g., packing)
  - JIT compiled code

- CFI is not perfect anyways

CFI Practicality: Coarse-Grained CFI

- Practical Control Flow Integrity and Randomization for Binary Executables, ***Oakland 2013***
- Control Flow Integrity for COTS binaries, ***USENIX Security 2013***
- Transparent ROP Exploit Mitigation Using Indirect Branch Tracing, ***USENIX Security 2013***
- ROPecker: A Generic and Practical Approach for Defending against ROP attacks, ***NDSS 2014***

Attacking Coarse-Grained CFI

- Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection, ***USENIX Security 2014***

- Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard, ***USENIX Security 2014***

- Out of Control: Overcoming Control-Flow Integrity, ***Oakland 2014***

Summary

- The CFI security policy dictates that software execution must follow a path of a Control-Flow Graph (CFG) determined ***ahead of time***.
- Type confusion bugs happen when a program misuses types
- ***Type confusion allows attackers to trigger memory corruption or disclosure***
- Use After Free is one of the major causes of type confusion

Question?