CSE 127: Computer Security

Stack Buffer Overflows

Deian Stefan
Slides adopted from Kirill Levchenko
Today

- Review: stack overflow attacks
- Shellcode
- Defenses
# Process Memory Layout

- **Stack**
- **Heap**
- **Data**
  - Static variables
- **Text**
  - Executable code

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapped</td>
</tr>
<tr>
<td>heap</td>
</tr>
<tr>
<td>.bss</td>
</tr>
<tr>
<td>.data</td>
</tr>
<tr>
<td>.text</td>
</tr>
</tbody>
</table>
The Stack

- Function local variables
- Function arguments
- Control state
The Stack

- Stack divided into **frames**
  - Frame stores locals and args to called functions

- **Stack pointer** points to
  - x86: Stack grows down (from high to low addresses)
  - x86: Stored in ESP register

- **Frame pointer** points to
  - Also called **base pointer**
  - x86: Stored in EBP register
The Stack

• Stack divided into **frames**
  ➤ Frame stores locals and args to called functions

• **Stack pointer** points to top of stack
  ➤ x86: Stack grows down (from high to low addresses)
  ➤ x86: Stored in **ESP** register

• **Frame pointer** points to caller’s stack frame
  ➤ Also called **base pointer**
  ➤ x86: Stored in **EBP** register
Function Call Example

```c
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```

Source: http://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/
_foobar:
; ebp must be preserved across calls. Since
; this function modifies it, it must be
; saved.
push ebp
; From now on, ebp points to the current stack
; frame of the function
mov ebp, esp
; Make space on the stack for local variables
sub esp, 16
; eax <-- a. eax += 2. then store eax in x
mov eax, DWORD PTR [ebp+8]
add eax, 2
mov DWORD PTR [ebp-4], eax
; eax <-- b. eax += 3. then store eax in yy
mov eax, DWORD PTR [ebp+12]
add eax, 3
mov DWORD PTR [ebp-8], eax
; eax <-- c. eax += 4. then store eax in zz
mov eax, DWORD PTR [ebp+16]
add eax, 4
mov DWORD PTR [ebp-12], eax
; add xx + yy + zz and store it in sum
mov eax, DWORD PTR [ebp-8]
mov edx, DWORD PTR [ebp-4]
lea eax, [edx+eax]
add eax, DWORD PTR [ebp-12]
mov DWORD PTR [ebp-16], eax
; Compute final result into eax
mov eax, DWORD PTR [ebp-4]
imul eax, DWORD PTR [ebp-8]
imul eax, DWORD PTR [ebp-12]
add eax, DWORD PTR [ebp-16]
; The leave instruction here is equivalent to:
;   mov esp, ebp ; pop ebp
leave
ret
Function Call Example

```c
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```

Source: http://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/
```c
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```
```c
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;
    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```

Source: http://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```c
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```
_foobarn:
  ; ebp must be preserved across calls. Since
  ; this function modifies it, it must be
  ; saved.
  ;
  push  ebp

  ; From now on, ebp points to the current stack
  ; frame of the function
  ;
  mov   ebp, esp

  ; Make space on the stack for local variables
  ;
  sub   esp, 16

  ; eax <-- a. eax += 2. then store eax in xx
  ;
  mov   eax, DWORD PTR [ebp+8]
  add   eax, 2
  mov   DWORD PTR [ebp-4], eax

  ; eax <-- b. eax += 3. then store eax in yy
  ;
  mov   eax, DWORD PTR [ebp+12]
  add   eax, 3
  mov   DWORD PTR [ebp-8], eax

  ; eax <-- c. eax += 4. then store eax in zz
  ;
  mov   eax, DWORD PTR [ebp+16]
  add   eax, 4
  mov   DWORD PTR [ebp-12], eax

Source: http://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/
```assembly
_foobar:
    ; ebp must be preserved across calls. Since
    ; this function modifies it, it must be
    ; saved.
    ;
    push ebp

    ; From now on, ebp points to the current stack
    ; frame of the function
    ;
    mov ebp, esp

    ; Make space on the stack for local variables
    ;
    sub esp, 16

    ; eax <-- a. eax += 2. then store eax in xx
    ;
    mov eax, DWORD PTR [ebp+8]
    add eax, 2
    mov DWORD PTR [ebp-4], eax

    ; eax <-- b. eax += 3. then store eax in yy
    ;
    mov eax, DWORD PTR [ebp+12]
    add eax, 3
    mov DWORD PTR [ebp-8], eax

    ; eax <-- c. eax += 4. then store eax in zz
    ;
    mov eax, DWORD PTR [ebp+16]
    add eax, 4
    mov DWORD PTR [ebp-12], eax
```

The diagram shows the stack frame with `ebp` pointing to the end of the main stack frame and `esp` pointing to the end of the local variables.

Source: [http://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/](http://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/)
_foobar:

; ebp must be preserved across calls. Since
; this function modifies it, it must be
; saved.
;
push ebp

; From now on, ebp points to the current stack
; frame of the function
;
mov ebp, esp

; Make space on the stack for local variables
;
sub esp, 16

; eax <-- a. eax += 2. then store eax in xx
;
mov eax, DWORD PTR [ebp+8]
add eax, 2
mov DWORD PTR [ebp-4], eax

; eax <-- b. eax += 3. then store eax in yy
;
mov eax, DWORD PTR [ebp+12]
add eax, 3
mov DWORD PTR [ebp-8], eax

; eax <-- c. eax += 4. then store eax in zz
;
mov eax, DWORD PTR [ebp+16]
add eax, 4
mov DWORD PTR [ebp-12], eax

ret

← ESP, EBP
_foobar:
; ebp must be preserved across calls. Since
; this function modifies it, it must be
; saved.
;
push ebp

; From now on, ebp points to the current stack
; frame of the function
;
    mov ebp, esp

; Make space on the stack for local variables
;
    sub esp, 16

; eax <-- a. eax += 2. then store eax in xx
;
    mov eax, DWORD PTR [ebp+8]
    add eax, 2
    mov DWORD PTR [ebp-4], eax

; eax <-- b. eax += 3. then store eax in yy
;
    mov eax, DWORD PTR [ebp+12]
    add eax, 3
    mov DWORD PTR [ebp-8], eax

; eax <-- c. eax += 4. then store eax in zz
;
    mov eax, DWORD PTR [ebp+16]
    add eax, 4
    mov DWORD PTR [ebp-12], eax

ret
_foobar:
    ; ebp must be preserved across calls. Since
    ; this function modifies it, it must be
    ; saved.
    ;
    push ebp

    ; From now on, ebp points to the current stack
    ; frame of the function
    ;
    mov ebp, esp

    ; Make space on the stack for local variables
    ;
    sub esp, 16

    ; eax <-- a. eax += 2. then store eax in xx
    ;
    mov eax, DWORD PTR [ebp+8]
    add eax, 2
    mov DWORD PTR [ebp-4], eax

    ; eax <-- b. eax += 3. then store eax in yy
    ;
    mov eax, DWORD PTR [ebp+12]
    add eax, 3
    mov DWORD PTR [ebp-8], eax

    ; eax <-- c. eax += 4. then store eax in zz
    ;
    mov eax, DWORD PTR [ebp+16]
    add eax, 4
    mov DWORD PTR [ebp-12], eax

Source: http://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/
_foobar:
    ; ebp must be preserved across calls. Since
    ; this function modifies it, it must be
    ; saved.
    ;
    push ebp

    ; From now on, ebp points to the current stack
    ; frame of the function
    ;
    mov ebp, esp

    ; Make space on the stack for local variables
    ;
    sub esp, 16

    ; eax <-- a. eax += 2. then store eax in xx
    ;
    mov eax, DWORD PTR [ebp+8]
    add eax, 2
    mov DWORD PTR [ebp-4], eax

    ; eax <-- b. eax += 3. then store eax in yy
    ;
    mov eax, DWORD PTR [ebp+12]
    add eax, 3
    mov DWORD PTR [ebp-8], eax

    ; eax <-- c. eax += 4. then store eax in zz
    ;
    mov eax, DWORD PTR [ebp+16]
    add eax, 4
    mov DWORD PTR [ebp-12], eax

Source: http://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/
mov DWORD PTR [ebp-8], eax

; eax <-- c. eax += 4. then store eax in zz
;
mov eax, DWORD PTR [ebp+16]
add eax, 4
mov DWORD PTR [ebp-12], eax

; add xx + yy + zz and store it in sum
;
mov eax, DWORD PTR [ebp-8]
mov edx, DWORD PTR [ebp-4]
lea eax, [edx+eax]
add eax, DWORD PTR [ebp-12]
mov DWORD PTR [ebp-16], eax

; Compute final result into eax, which
; stays there until return
;
mov eax, DWORD PTR [ebp-4]
imul eax, DWORD PTR [ebp-8]
imul eax, DWORD PTR [ebp-12]
add eax, DWORD PTR [ebp-16]

; The leave instruction here is equivalent to:
;
mov esp, ebp
pop ebp
;
Which cleans the allocated locals and restores
ebp.
;
leave
ret
mov  DWORD PTR [ebp-8], eax

; eax <-- c. eax += 4. then store eax in zz
;
mov  eax, DWORD PTR [ebp+16]
add  eax, 4
mov  DWORD PTR [ebp-12], eax

; add xx + yy + zz and store it in sum
;
mov  eax, DWORD PTR [ebp-8]
mov  edx, DWORD PTR [ebp-4]
lea  eax, [edx+eax]
add  eax, DWORD PTR [ebp-12]
mov  DWORD PTR [ebp-16], eax

; Compute final result into eax, which
; stays there until return
;
mov  eax, DWORD PTR [ebp-4]
imul eax, DWORD PTR [ebp-8]
imul eax, DWORD PTR [ebp-12]
add  eax, DWORD PTR [ebp-16]

; The leave instruction here is equivalent to:
;
; mov esp, ebp
; pop ebp
;
; Which cleans the allocated locals and restores
; ebp.
;
leave
ret

← EBP, ESP

| c |
| b |
| a |
| ret |
| sfp |
| xx |
| yy |
| zz |
| sum |
mov DWORD PTR [ebp-8], eax

; eax <-- c. eax += 4. then store eax in zz
;
mov eax, DWORD PTR [ebp+16]
add eax, 4
mov DWORD PTR [ebp-12], eax

; add xx + yy + zz and store it in sum
;
mov eax, DWORD PTR [ebp-8]
mov edx, DWORD PTR [ebp-4]
lea eax, [edx+eax]
add eax, DWORD PTR [ebp-12]
mov DWORD PTR [ebp-16], eax

; Compute final result into eax, which
; stays there until return
;
mov eax, DWORD PTR [ebp-4]
imul eax, DWORD PTR [ebp-8]
imul eax, DWORD PTR [ebp-12]
add eax, DWORD PTR [ebp-16]

; The leave instruction here is equivalent to:
;
    mov esp, ebp
    pop ebp
;
; Which cleans the allocated locals and restores
; ebp.
;
leave
ret

Source: http://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/
mov    DWORD PTR [ebp-8], eax

; eax = c. eax += 4. then store eax in zz
;
mov    eax, DWORD PTR [ebp+16]
add    eax, 4
mov    DWORD PTR [ebp-12], eax

; add xx + yy + zz and store it in sum
;
mov    eax, DWORD PTR [ebp-8]
mov    edx, DWORD PTR [ebp-4]
lea    eax, [edx+eax]
add    eax, DWORD PTR [ebp-12]
mov    DWORD PTR [ebp-16], eax

; Compute final result into eax, which
; stays there until return
;
mov    eax, DWORD PTR [ebp-4]
imul   eax, DWORD PTR [ebp-8]
imul   eax, DWORD PTR [ebp-12]
add    eax, DWORD PTR [ebp-16]

; The leave instruction here is equivalent to:
;
;     mov esp, ebp
;     pop ebp
;
; Which cleans the allocated locals and restores
; ebp.
;
leave
ret

← EBP  points to end of main stack frame

← ESP  

  c
  b
  a
  ret
  sfp
With Buffers

example1.c:

```c
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}

void main() {
    function(1,2,3);
}
```

(source: AlephOne)
• strcpy will copy memory from str to buffer

• So?

```
buffer sfp ret *str
[ ][ ][ ][ ]
```

example2.c
```
void function(char *str) {
    char buffer[16];
    strcpy(buffer,str);
}

void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```

(source: AlephOne)
With Buffers

- `strcpy` will copy memory from `str` to `buffer` until `\0`.
- **So?**
  - if length of string longer than `buffer`, `strcpy` will copy string over `sfp` and `ret`.

```
example2.c  
-------------
void function(char *str) {
    char buffer[16];
    strcpy(buffer,str);
}

void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
  
  (source: AlephOne)
```
Stack Buffer Overflow

• If source string of `strcpy` controlled by attacker (and destination is on the stack)
  ➢ Attacker gets to control where the function returns by overwriting `ret`
  ➢ Attacker gets to transfer control to anywhere!

• Where do you jump?
Taking Control

• Let’s jump to code that does what we want

• **Where?** We have control of string!
  
  ➤ Put code in string
  ➤ Jump to start of string
Taking Control

• Let’s jump to code that does what we want

• **Where?** We have control of string!
  ➤ Put code in string
  ➤ Jump to start of string

```
<table>
<thead>
<tr>
<th>bottom of memory</th>
<th>DDDDDDDDEEEEEE</th>
<th>EEEE</th>
<th>FFFF</th>
<th>FFFF</th>
<th>FFFF</th>
<th>FFFF</th>
<th>top of memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>89ABabcdef0123456789AB</td>
<td>CDEF</td>
<td>0123</td>
<td>4567</td>
<td>89AB</td>
<td>CDEF</td>
<td>memory</td>
</tr>
<tr>
<td></td>
<td>sfp</td>
<td>ret</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[SSSSSSSSSSSSSSSSSSSSSSSSSS]</td>
<td>[SSSS]</td>
<td>[0xD8]</td>
<td>[0x01]</td>
<td>[0x02]</td>
<td>[0x03]</td>
<td></td>
</tr>
<tr>
<td>top of stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bottom of stack</td>
</tr>
</tbody>
</table>
```
Shellcode

- **Shellcode**: small code fragment that receives initial control in an control flow hijack exploit
  - Control flow hijack: taking control of instruction ptr
- Earliest attacks used shellcode to exec a shell
  - Target a setuid root program, gives you root shell
Shellcode

```c
void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```
Shellcode

• Can we just take output from gcc/clang?
  ➤ A: yes B: no
Shellcode

- There are some restrictions
  - Shellcode cannot contain null characters ‘\0’
  - Why not?
  - Recipe:
    a) Have the null terminated string "/bin/sh" somewhere in memory.
    b) Have the address of the string "/bin/sh" somewhere in memory followed by a null long word.
    c) Copy 0xb into the EAX register.
    d) Copy the address of the address of the string "/bin/sh" into the EBX register.
    e) Copy the address of the string "/bin/sh" into the ECX register.
    f) Copy the address of the null long word into the EDX register.
    g) Execute the int $0x80 instruction.
Why does this not really work?
How can we address this?

<table>
<thead>
<tr>
<th>bottom of memory</th>
<th>DDDDDDDDEEEEEE</th>
<th>EEEE</th>
<th>FFFF</th>
<th>FFFF</th>
<th>FFFF</th>
<th>FFFF</th>
<th>FFFF</th>
<th>top of memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory</td>
<td>89ABCDEF0123456789AB</td>
<td>CDEF</td>
<td>0123</td>
<td>4567</td>
<td>89AB</td>
<td>CDEF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>buffer</td>
<td>sfp</td>
<td>ret</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- [ ]
- [ ]
- [ ]
- [ ]
- [ ]
- [ ]
- [ ]
- [ ]

<table>
<thead>
<tr>
<th>top of stack</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>bottom of stack</th>
</tr>
</thead>
</table>
How can we address this?
How can we address this?
Metasploit to the rescue!

```bash
msf payload(shell_bind_tcp) > generate -h
Usage: generate [options]

Generates a payload.

OPTIONS:
- `-E` Force encoding.
- `-b` The list of characters to avoid: `\x00\xff`
- `-e` The name of the encoder module to use.
- `-f` The output file name (otherwise stdout)
- `-h` Help banner.
- `-i` the number of encoding iterations.
- `-k` Keep the template executable functional
- `-o` A comma separated list of options in VAR=VAL format.
- `-p` The Platform for output.
- `-s` NOP sled length.
- `-t` The output format: raw,ruby,rb,perl,pl,c,js_be,js_le,java,dll,exe,exe-small,elf,macho,vba,etc
- `-x` The executable template to use
```
Buffer Overflow Defenses

- Avoid unsafe functions
- Stack canary
- Separate control stack
- Address Space Layout Randomization (ASLR)
- Memory writable or executable, not both ($W^X$)
- Control flow integrity (CFI)
Avoiding Unsafe Functions

- strcpy, strcat, gets, etc.

- **Plus:** Good idea in general

- **Minus:** Requires manual code rewrite

- **Minus:** Non-library functions may be vulnerable
  - E.g. user creates their own strcpy

- **Minus:** No guarantee you found everything

- **Minus:** alternatives are also error-prone
Even printf is tricky

If buf is under control of attacker
is: printf(“%s\n”, buf) safe?

A: yes, B: no
Even printf is tricky

If buf is under control of attacker is: printf(buf) safe?

A: yes, B: no
Even printf is tricky

Is printf(“%s
) safe?

A: yes, B: no
Even printf is tricky

printf can be used to read and write memory

control flow hijacking!

Exploiting Format String Vulnerabilities

scut / team teso

September 1, 2001

https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf
Buffer Overflow Defenses

- Avoid unsafe functions
- Stack canary
- Separate control stack
- Address Space Layout Randomization (ASLR)
- Memory writable or executable, not both ($W^X$)
- Control flow integrity (CFI)
Stack Canary

- Special value placed before return address
  - Secret random value chosen at program start
  - String terminator ‘\0’
- Gets overwritten during buffer overflow
- Check canary before jumping to return address
- Automatically inserted by compiler
  - GCC: -fstack-protector or -fstack-protector-strong
Stack Canary

• **Plus:** No code changes required, only recompile

• **Minus:**

• **Minus:**

• **Minus:**
Stack Canary

• **Plus:** No code changes required, only recompile

• **Minus:** Performance penalty per return

• **Minus:** Only protects against stack smashing

• **Minus:** Fails if attacker can read memory
Buffer Overflow Defenses

- Avoid unsafe functions
- Stack canary
- Separate control stack
- Address Space Layout Randomization (ASLR)
- Memory writable or executable, not both (W^X)
- Control flow integrity (CFI)
“SafeStack is an instrumentation pass that protects programs against attacks based on stack buffer overflows, without introducing any measurable performance overhead. It works by separating the program stack into two distinct regions: the safe stack and the unsafe stack. The safe stack stores return addresses, register spills, and local variables that are always accessed in a safe way, while the unsafe stack stores everything else. This separation ensures that buffer overflows on the unsafe stack cannot be used to overwrite anything on the safe stack.”
Address Space Layout Randomization

• Change location of stack, heap, code, static vars
• Works because attacker needs address of shellcode
• Layout must be unknown to attacker
  ➢ Randomize on every launch (best)
  ➢ Randomize at compile time
• Implemented on most modern OSes in some form
Traditional Memory Layout

- Stack
- mapped
- heap
- .bss
- .data
- .text
PaX Memory Layout

random stack base

random base

random base
Address Space Layout Randomization

- **Plus:** No code changes or recompile required
- **Minus:** 32-bit arch get limited protection
- **Minus:**
- **Minus:**
- **Minus:**
Address Space Layout Randomization

- **Plus:** No code changes or recompile required
- **Minus:** 32-bit arch get limited protection
- **Minus:** Fails if attacker can read memory
- **Minus:** Load-time overhead
- **Minus:** No exec img sharing between processes
W^X: write XOR execute

- Use MMU to ensure memory cannot be both writeable and executable at same time
- Code segment: executable, not writeable
- Stack, heap, static vars: writeable, not executable
- Supported by most modern processors
- Implemented by modern operating systems
W^X: write XOR execute

• **Plus:** No code changes or recompile required

• **Minus:** Requires hardware support

• **Minus:**

• **Minus:**
W^X: write XOR execute

- **Plus:** No code changes or recompile required
- **Minus:** Requires hardware support
- **Minus:** Defeated by return-oriented programming
- **Minus:** Does not protect JITed code
Buffer Overflow Defenses

- Avoid unsafe functions
- Stack canary
- Separate control stack
- Address Space Layout Randomization (ASLR)
- Memory writable or executable, not both ($W^X$)
- Control flow integrity (CFI)
Control Flow Integrity

• Check destination of every indirect jump
  ➢ Function returns
  ➢ Function pointers
  ➢ Virtual methods

• What are the valid destinations?
  ➢
  ➢
Control Flow Integrity

• Check destination of every indirect jump
  ➤ Function returns
  ➤ Function pointers
  ➤ Virtual methods

• What are the valid destinations?
  ➤ Caller of every function known at compile time
  ➤ Class hierarchy limits possible virtual function instances
CFI

- **Plus:** No code changes or hardware support
- **Plus:** Protects against many vulnerabilities
- **Minus:**
- **Minus:**
- **Minus:**
CFI

- **Plus**: No code changes or hardware support
- **Plus**: Protects against many vulnerabilities
- **Minus**: Performance overhead
- **Minus**: Requires smarter compiler
- **Minus**: Requires having all code available