University of BRISTOL

Systems Security

COMSM1500

bristol.ac.uk

Buffer overflow

University of BRISTOL

Assembly language

Small refresher

bristol.ac.uk

_start

```
.text                       # code segment
    .global _start          # export

_start:
    movl $0x01, %eax        # exit
    movl $0x00, %ebx        # return code
    int  $0x80              # syscall
```

hello world

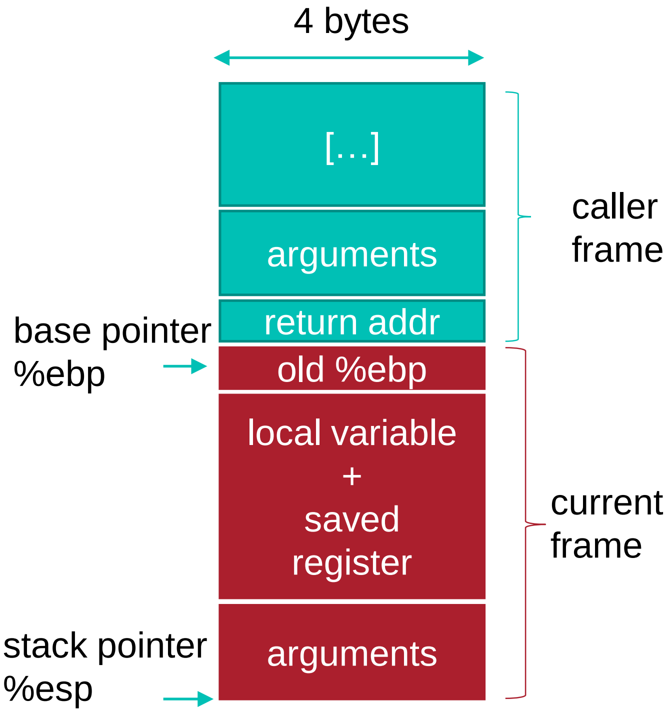
```
# int write(int fd, char* buf, int len)

   movl $0x04, %eax      # write
   movl $0x01, %ebx      # stdout
   movl $str,  %ecx      # buffer
   movl $14,   %edx      # length
   int  $0x80

.data
   str:
   .ascii "Hello, World!\n"
```

Stack

Small refresher

The stack

- Current stack frame ("top" to bottom)
  - arguments for function about to be called
  - saved register context (if reused)
  - local variable
  - old base pointer
- Caller's stack frame
  - return address
    - pushed by call instruction
  - Arguments for this call



Example

```
int zip1 = 15213;
int zip2 = 98915;

void call_swap() {
    swap(&zip1, &zip2);
}

void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Example

```c
int zip1 = 15213;
int zip2 = 98915;

void call_swap() {
    swap(&zip1, &zip2);
}
```

Example

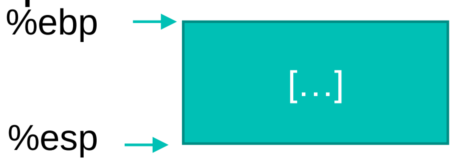
```
int zip1 = 15213;
int zip2 = 98915;

void call_swap() {
    swap(&zip1, &zip2);
}
```

```
# void call_swap()
…
pushl $zip1
pushl $zip2
call swap
…
```

Example

```
int zip1 = 15213;
int zip2 = 98915;

void call_swap() {
    swap(&zip1, &zip2);
}
```

```
# void call_swap()
…
pushl $zip1
pushl $zip2
call swap
…
```

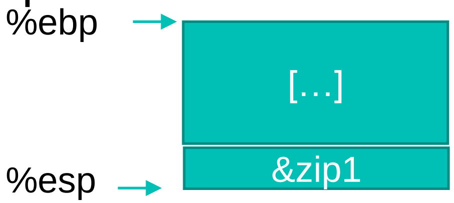


Example

- int zip1 = 15213;
- int zip2 = 98915;

- **void** call_swap() {
-     swap(&zip1, &zip2);
- }

- *# void call_swap()*
- …
- pushl $zip1
- pushl $zip2
- call swap
- …

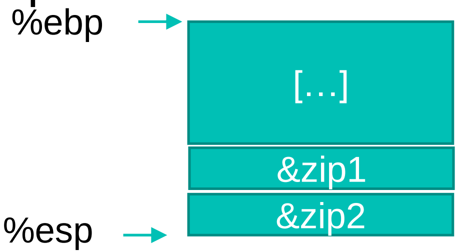


Example

- `int zip1 = 15213;`
- `int zip2 = 98915;`

- **`void`** `call_swap() {`
- `    swap(&zip1, &zip2);`
- `}`

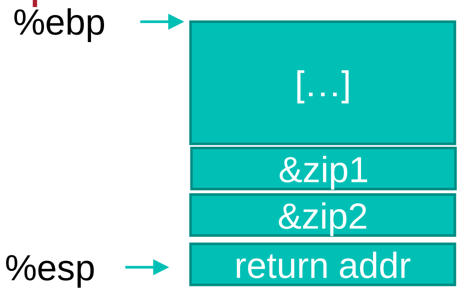
- *`# void call_swap()`*
- …
- `pushl $zip1`
- `pushl $zip2`
- `call swap`
- …



Example

```
int zip1 = 15213;
int zip2 = 98915;

void call_swap() {
    swap(&zip1, &zip2);
}
```

```
# void call_swap()
…
pushl $zip1
pushl $zip2
call swap
…
```


%ebp
[…]
&zip1
&zip2
%esp
return addr

Example

```
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0
}
```

```
# void swap(int *xp, int *yp)
```

```
pushl %ebp
movl %esp, %ebp
pushl %ebx
```

Set up

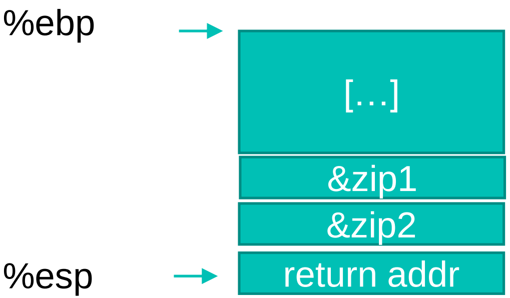
```
movl 12(%ebp), %ecx
movl 8(%ebp), %edx
movl (%ecx), %eax
movl (%edx); %ebx
movl %eax, (%edx)
movl %ebx, (%ecx)
```

body

```
movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret
```

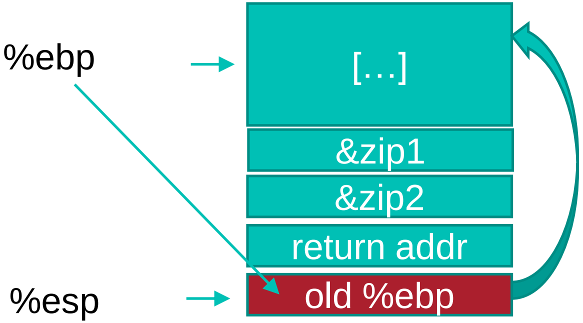
finish

Example



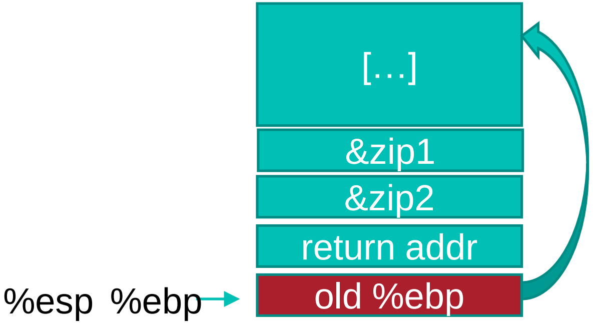
- `pushl %ebp`
- `movl %esp, %ebp`
- `pushl %ebx`

Example



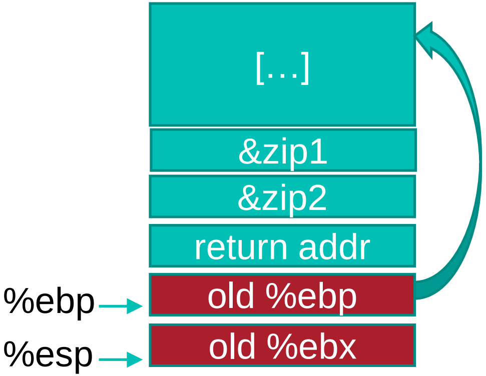
- `pushl %ebp`
- `movl %esp, %ebp`
- `pushl %ebx`

Example



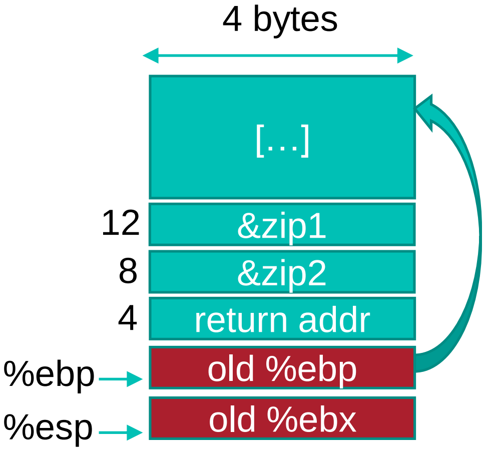
```
▪ pushl %ebp
▪ movl %esp, %ebp
▪ pushl %ebx
```

Example



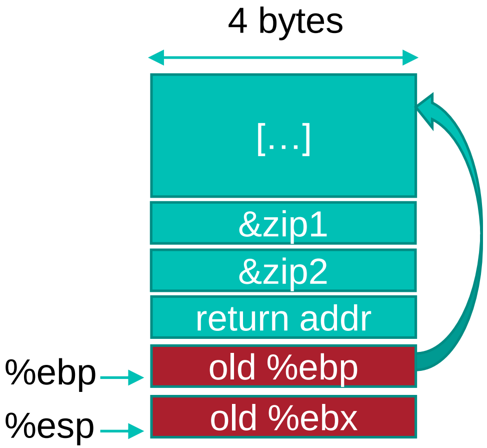
- `pushl %ebp`
- `movl %esp, %ebp`
- `pushl %ebx`

Example



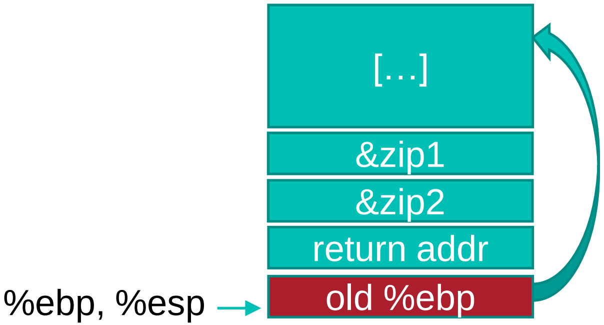
- `movl 12(%ebp), %ecx`
- `movl 8(%ebp), %edx`
- …

Example



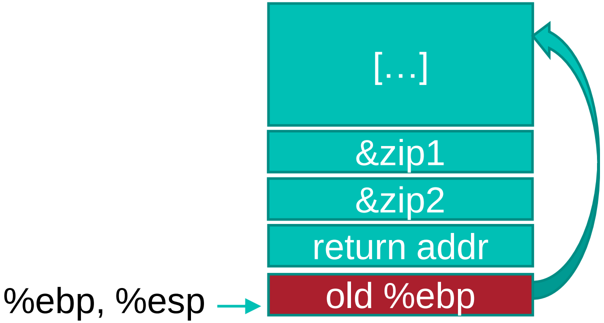
- movl -4(%ebp), %ebx
- movl %ebp, %esp
- popl %ebp
- ret

Example



```
▪ movl -4(%ebp), %ebx
▪ movl %ebp, %esp
▪ popl %ebp
▪ ret
```

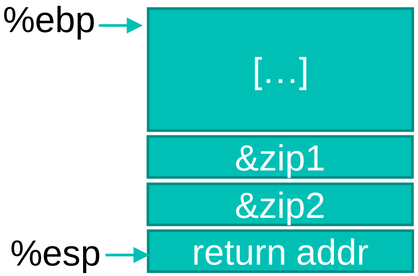
Example



- movl -4(%ebp), %ebx
- movl %ebp, %esp
- popl %ebp
- ret

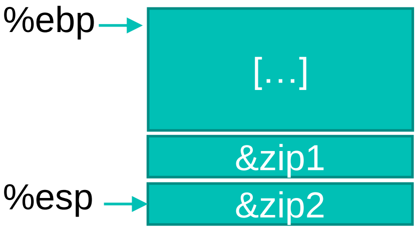
popl %ebx

Example



- `movl -4(%ebp), %ebx`
- `movl %ebp, %esp`
- `popl %ebp`
- `ret`

Example



- `movl -4(%ebp), %ebx`
- `movl %ebp, %esp`
- `popl %ebp`
- `ret`

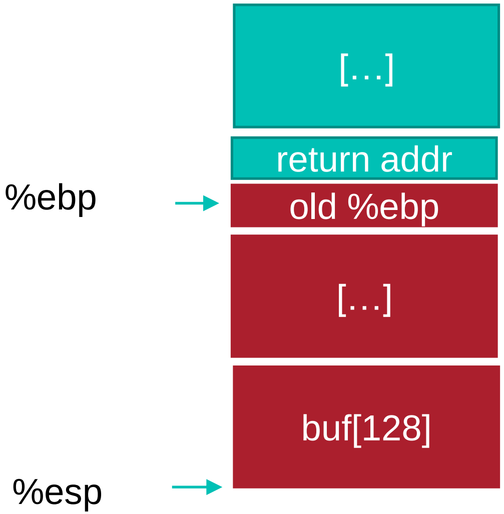
University of BRISTOL

Buffer overflow

bristol.ac.uk

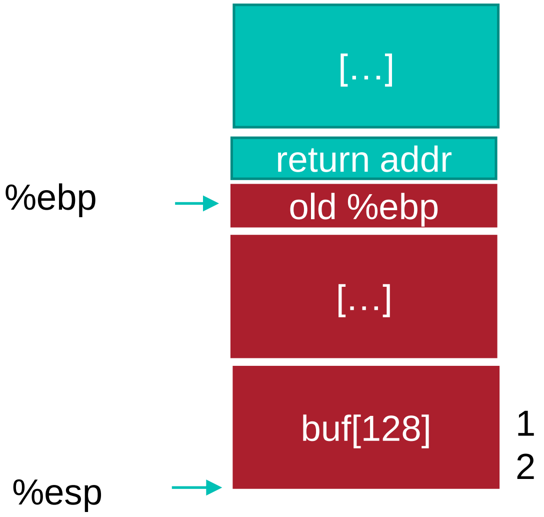
Example

```
int read_get(void) {
  char buf[128];
  int i;
  gets(buf);
  i = atoi(buf);
  return i;
}

int main() {
  x = read_get();
  printf(“%s”, x);
}
```



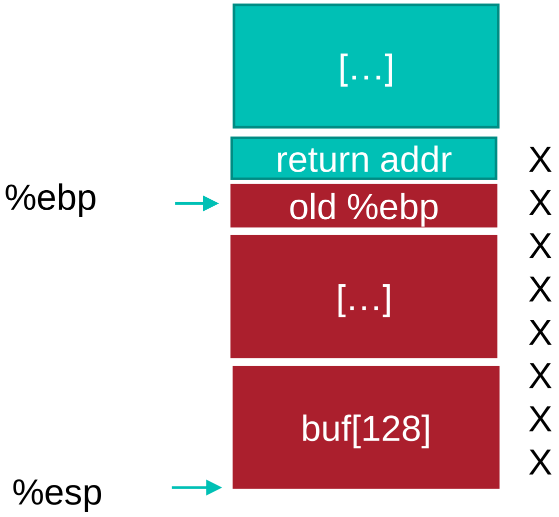
Example

```
int read_get(void) {
  char buf[128];
  int i;
  gets(buf);
  i = atoi(buf);
  return i;
}

int main() {
  x = read_get();
  printf(“%s”, x);
}
```



Example

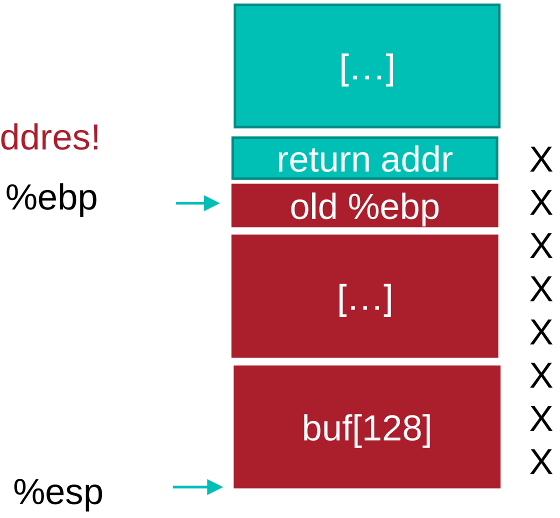
```
int read_get(void) {
  char buf[128];
  int i;
  gets(buf);
  i = atoi(buf);
  return I;
}

int main() {
  x = read_get();
  printf(“%s”, x);
}
```



Example

```
int read_get(void) {
  char buf[128];
  int i;
  gets(buf);
  i = atoi(buf);
  return I;
}

int main() {
  x = read_get();
  printf(“%s”, x);
}
```

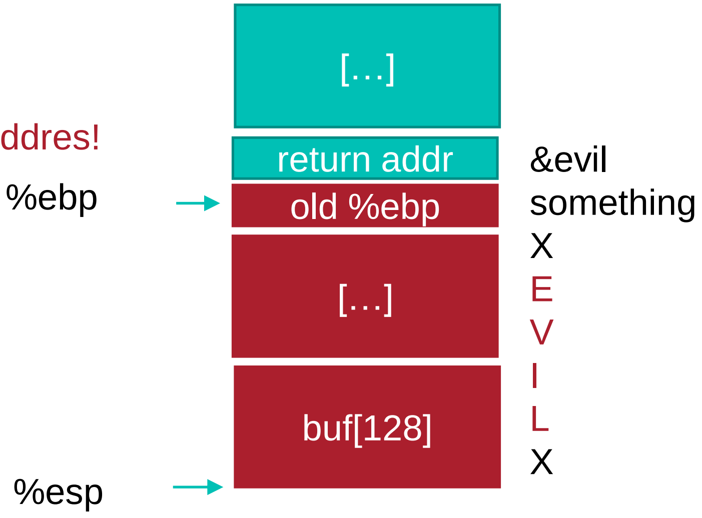
Changed returned addres!
and old ebp.



Example

```
int read_get(void) {
   char buf[128];
   int i;
   gets(buf);
   i = atoi(buf);
   return I;
}

int main() {
   x = read_get();
   printf(“%s”, x);
}
```

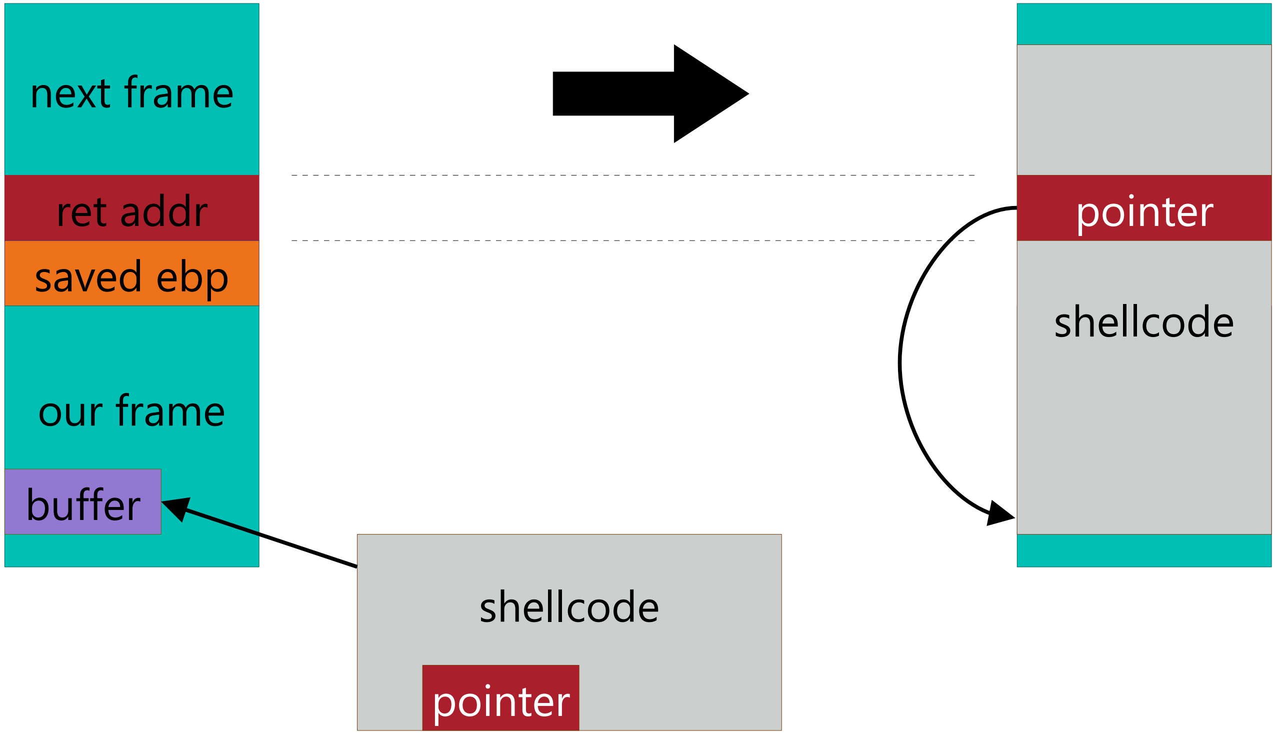
Changed returned addres!
and old ebp.



buffer overflows

- Vulnerability in C / assembly programs where the compiler does not enforce array bounds.
    - a[1]
    - *(a+1)
- Take over a setuid program, get root.

stack overflow



execve

- **NAME**
- execve - execute program

- **SYNOPSIS**
- **#include <unistd.h>**

- **int execve(const char** *filename,
- **char *const** argv[],
- **char *const** envp[]);

- **DESCRIPTION**
- **execve()** executes the program
- pointed to by filename.

execve in assembly

```
.section .data
    cmd: .asciz "/bin/sh"
    ptr: .int cmd
         .int 0

.section .text
    .globl _start
_start:
    mov $0x0b, %eax     # execve
    mov $cmd,  %ebx     # command
    mov $ptr,  %ecx     # args
    mov $0,    %edx     # env
    int $0x80
```

What’s the problem?

The problem

- mov $0x0b, %eax = B8 0B 00 00 00
  - B8: mov IMM32, %eax
  - those null bytes will terminate a strcpy/scanf/gets etc.
- challenge is to create shellcode with only “legal” bytes
- also, how to address your payload?
- For you to figure out in the coursework ;-)

University of BRISTOL

How to protect from this?

countermeasures

prevent

detect

recover

Prevent!

- Solution A: Avoid bugs in your C code!

Prevent!

- Solution A: Avoid bugs in your C code!
  - Maybe can check usage of problematic C functions?

Prevent!

- Solution A: Avoid bugs in your C code!
  - Maybe can check usage of problematic C functions?
  - What about raw pointer manipulations?

Prevent!

- Solution A: Avoid bugs in your C code!
  - Maybe can check usage of problematic C functions?
  - What about raw pointer manipulations?
  - Look at a real large C projects… does not look easy

Prevent!

- Solution A: Avoid bugs in your C code!
- Solution B: build tools

Prevent!

- Solution A: Avoid bugs in your C code!
- Solution B: build tools
  - To help find bugs

Prevent!

- Solution A: Avoid bugs in your C code!
- Solution B: build tools
  - To help find bugs
  - Static analysis

Prevent!

- Solution A: Avoid bugs in your C code!
- Solution B: build tools
  - To help find bugs
  - Static analysis

```
void foo(int *p) {
  int off;
  *z = p + off;
  if (off > 8)
    bar(8);
}
```

Prevent!

- Solution A: Avoid bugs in your C code!
- Solution B: build tools
  - To help find bugs
  - Static analysis

```
void foo(int *p) {
  int off;NOT INITIALIZED
  *z = p + off;
  if (off > 8)
    bar(8);
}
```

Prevent!

- Solution A: Avoid bugs in your C code!
- Solution B: build tools
  - To help find bugs
  - Static analysis

```
void foo(int *p) {
    int off;NOT INITIALIZED
    *z = p + off;
    if (off > 8) PROPAGATE ASSUMPTION ABOUT
        bar(off);  off VALUE
}
```

Prevent!

- Solution A: Avoid bugs in your C code!
- Solution B: build tools
  - To help find bugs
  - Static analysis
  - Fuzzing
    - Pushing massive amount of random value to a program
    - See if it crashes

Prevent!

- Solution A: Avoid bugs in your C code!
- Solution B: build tools
  - To help find bugs
  - Static analysis
  - Fuzzing
    - Pushing massive amount of random value to a program
    - See if it crashes
    - Can be a bit smarter and make sure we reach every branch in the program

Prevent!

- Solution A: avoid bugs in your C code!
- Solution B: build tools
- Solution C: use a memory safe language

Prevent!

- Solution A: avoid bugs in your C code!
- Solution B: build tools
- Solution C: use a memory safe language
  - JAVA, C#, Rust etc…

Prevent!

- Solution A: avoid bugs in your C code!
- Solution B: build tools
- Solution C: use a memory safe language
  - JAVA, C#, Rust etc…
  - Legacy code! (that's how the real world exists)

Prevent!

- Solution A: avoid bugs in your C code!
- Solution B: build tools
- Solution C: use a memory safe language
  - JAVA, C#, Rust etc…
  - Legacy code! (that's how the real world exists)
  - Need low level hardware access?

Prevent!

- Solution A: avoid bugs in your C code!
- Solution B: build tools
- Solution C: use a memory safe language
  - JAVA, C#, Rust etc…
  - Legacy code! (that’s how the real world exists)
  - Need low level hardware access?
  - Performance?

Prevent!

- Solution A: avoid bugs in your C code!
- Solution B: build tools
- Solution C: use a memory safe language
  - JAVA, C#, Rust etc…
  - Legacy code! (that’s how the real world exists)
  - Need low level hardware access?
  - Performance?
    - ➢ It used to be a problem, not necessarily anymore

Prevent!

- Solution A: avoid bugs in your C code!
- Solution B: build tools
- Solution C: use a memory safe language
  - JAVA, C#, Rust etc…
  - Legacy code! (that’s how the real world exists)
  - Need low level hardware access?
  - Performance?
    - It used to be a problem, not necessarily anymore
    - Is your program CPU bound anyway?

Prevent!

- Solution A: avoid bugs in your C code!
- Solution B: build tools
- Solution C: use a memory safe language

countermeasures

prevent

detect

recover

Next lecture

Detect!

Thank you

Office MVB 3.26

bristol.ac.uk