University of BRISTOL

# Systems Security

## COMSM1500

bristol.ac.uk

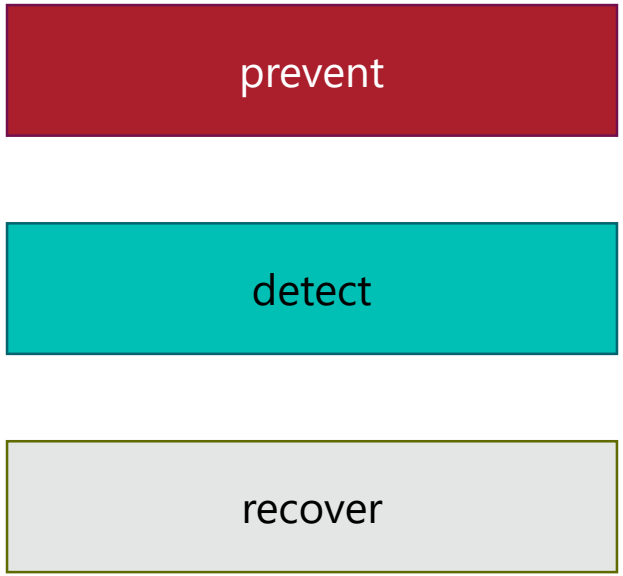
University of BRISTOL

# Buffer overflow

Continued…

bristol.ac.uk

# countermeasures



University of BRISTOL

# Detecting

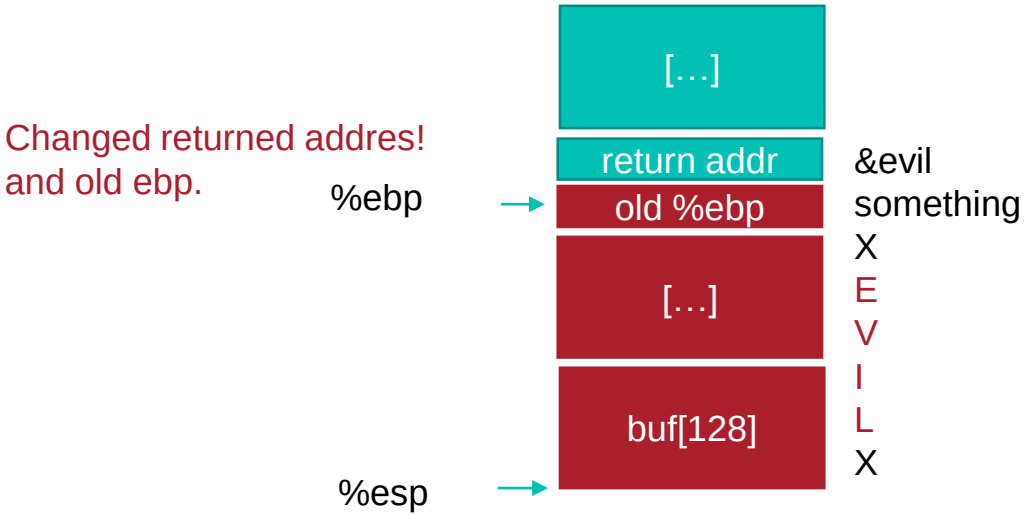
## Buffer overflow

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);
}
```



# Buffer overflow exploit

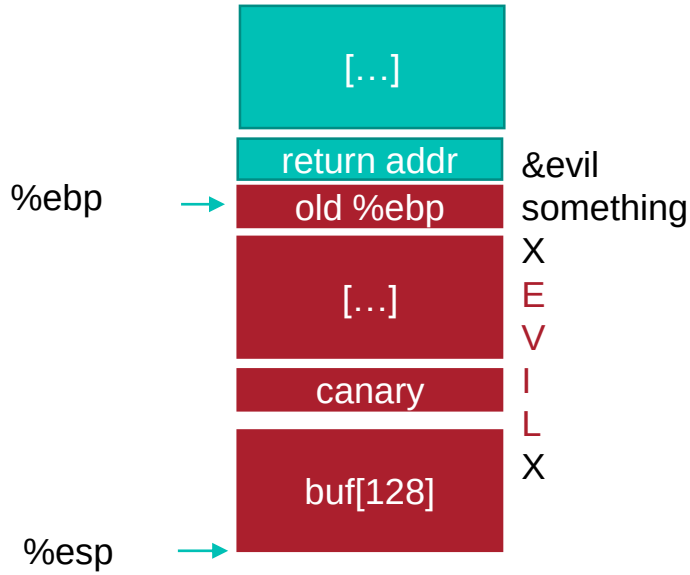
- Gaining control over the instruction pointer
  - i.e. changing return address
  - control what will be executed
- Make that pointer points to malicious code
  - embedding code (e.g. shell code last time)
  - jumping to unexpected part of code (i.e. open door)
- Gain control over stack pointer
  - i.e. control data

# Stack canaries

- Let attacker overwrite stack

# Stack canaries

- Let attacker overwrite stack
- Before return
  - Check the value of the canary
  - If it changed something bad happened
  - Compiler support



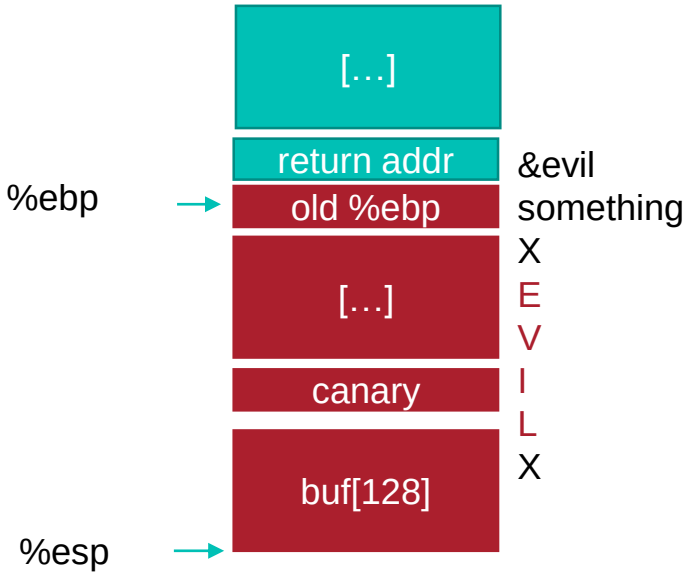
University of BRISTOL

# Problem?

bristol.ac.uk

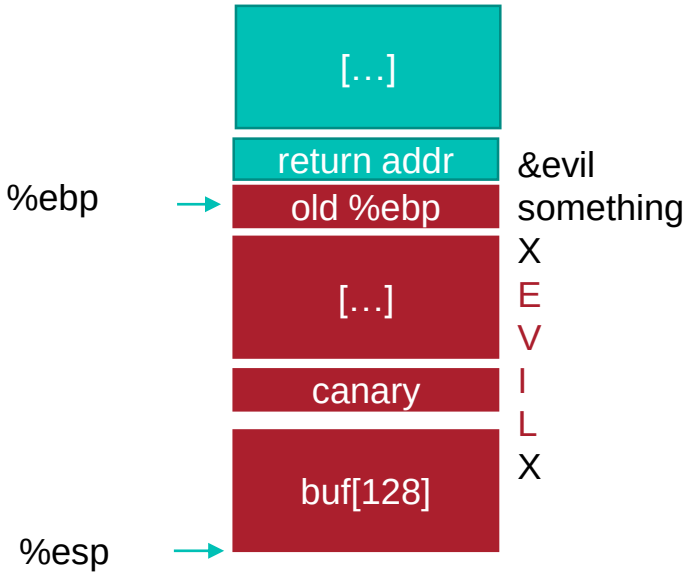
# Stack canaries

- Let attacker overwrite stack
- Before return
- Careful about canary value
  - if deterministic can be guessed and avoided



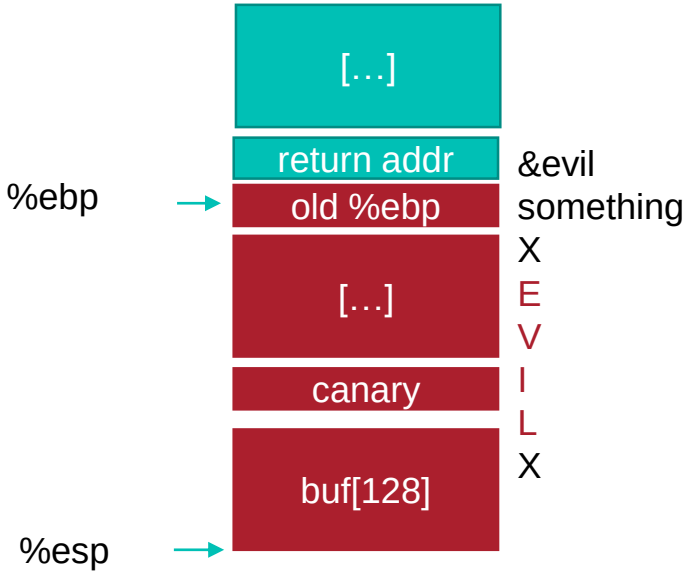
# Stack canaries

- Let attacker overwrite stack
- Before return
- Careful about canary value
  - if deterministic can be guessed and avoided
- Use some special characters
  - e.g. \0, EOF etc…
  - remember last week
  - would only work for some input functions



# Stack canaries

- Let attacker overwrite stack
- Before return
- Careful about canary value
  - if deterministic can be guessed and avoided
- Use some special characters
  - e.g. \0, EOF etc…
  - remember last week
  - would only work for some input functions
- Use some random value
  - careful with entropy



# When do canaries fail?

- When attacker overwrite function pointers

# When do canaries fail?

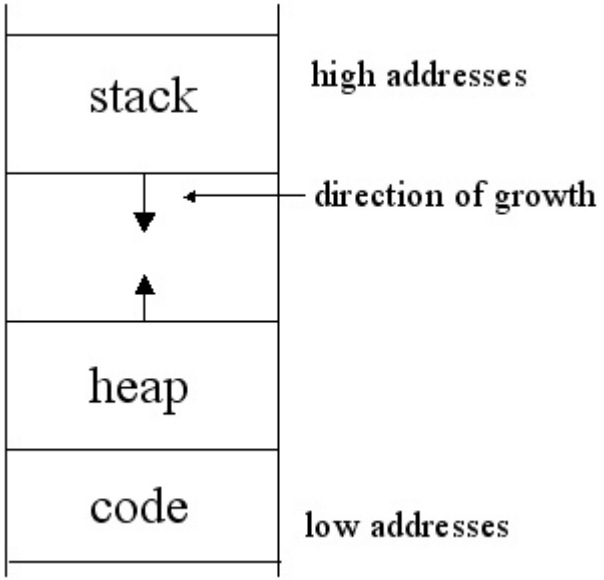
- When attacker overwrite function pointers
  - int *ptr = … ;
  - char buf[128];
  - gets(buff);
  - ptr(…);

# When do canaries fail?

- When attacker overwrite function pointers
- Can attacker guess the randomness?
  - Source of randomness is a research topics on its own!

# When do canaries fail?

- When attacker overwrite function pointers
- Can attacker guess the randomness?
- malloc and free (heap)
  - char *p, *q;
  - p = malloc(127);
  - q = malloc(127);
  - strcpy(p, buf);
  - free(p);
  - free(q);

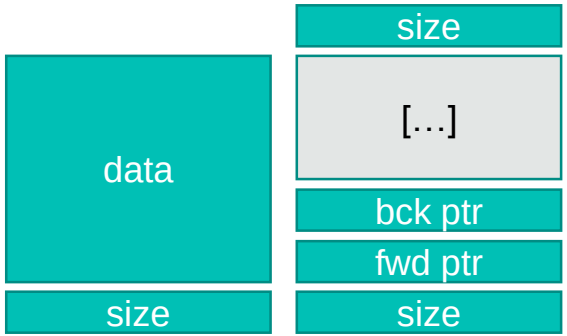


# When do canaries fail?

- When attacker overwrite function pointers
- Can attacker guess the randomness?
- malloc and free (heap)
  - char *p, *q;
  - p = malloc(127);
  - q = malloc(127);
  - strcpy(p, buf);
  - free(p);
  - free(q);

# When do canaries fail?

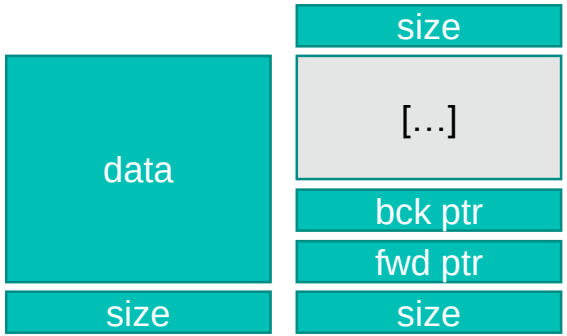
- When attacker overwrite function pointers
- Can attacker guess the randomness?
- malloc and free (heap)
  - char *p, *q;
  - p = malloc(127);
  - q = malloc(127);
  - strcpy(p, buf);
  - free(p);
  - free(q);




pointer and size for book keeping

# When do canaries fail?

- When attacker overwrite function pointers
- Can attacker guess the randomness?
- malloc and free (heap)
  - p = get_free_block(size);
  - bck = p->bck;
  - fwd = p->fwd;
  - fwd->bck = bck;
  - fwd->fwd = fwd;

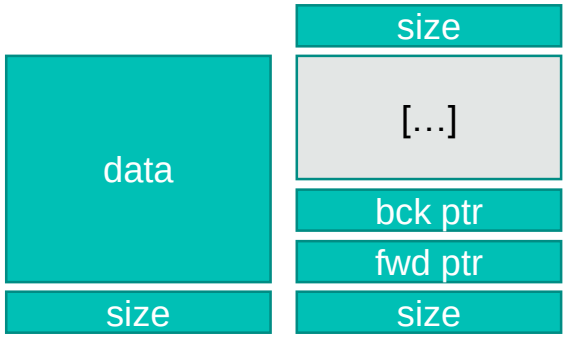



pointer and size for book keeping

# When do canaries fail?

- When attacker overwrite function pointers
- Can attacker guess the randomness?
- malloc and free (heap)
  - p = get_free_block(size);
  - bck = p->bck;
  - fwd = p->fwd;
  - fwd->bck = bck;
  - fwd->fwd = fwd;

GAINED CONTROL OF MEMORY ALLOCATION



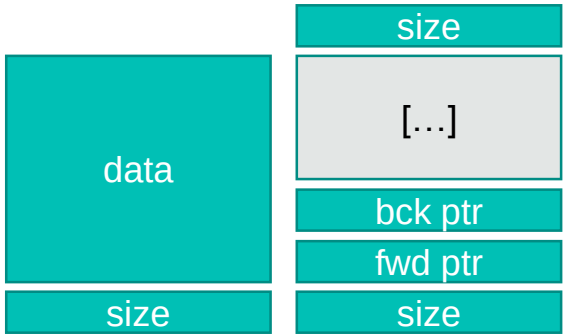

pointer and size for book keeping

# When do canaries fail?

Homework/exam question: Describe the canary technique to protect from buffer overflow exploits and discuss its limitation.

- When attacker overwrite function pointers
- Can attacker guess the randomness?
- malloc and free (heap)
  - p = get_free_block(size);
  - bck = p->bck;
  - fwd = p->fwd;
  - fwd->bck = bck;
  - fwd->fwd = fwd;

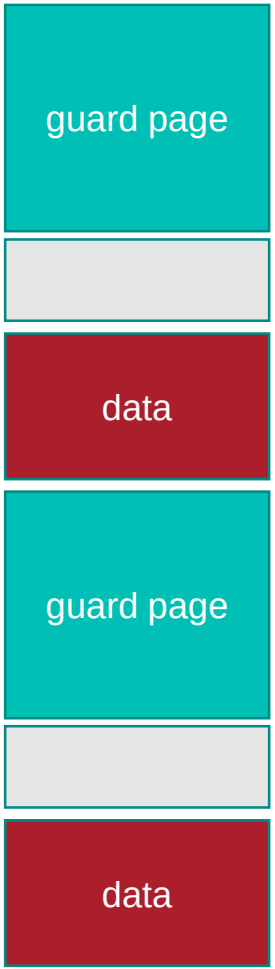
GAINED CONTROL OF MEMORY ALLOCATION




pointer and size for book keeping

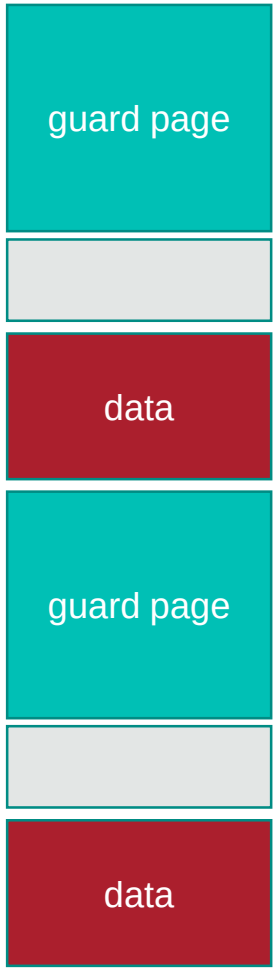
# Electric Fence

- Use guard page
  - Page with memory protection so that if touched, create a fault



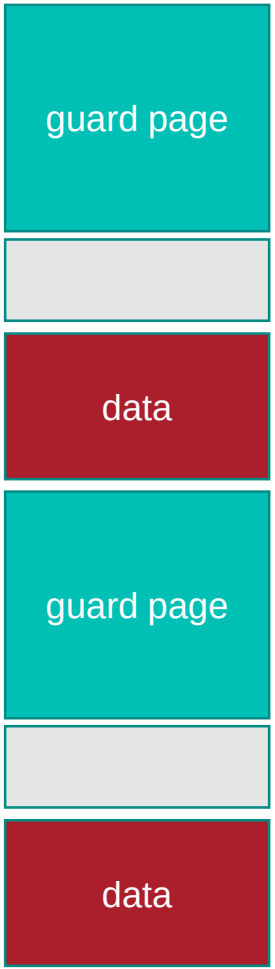
# Electric Fence

- Use guard page
  - Page with memory protection so that if touched, create a fault
- Fault immediate



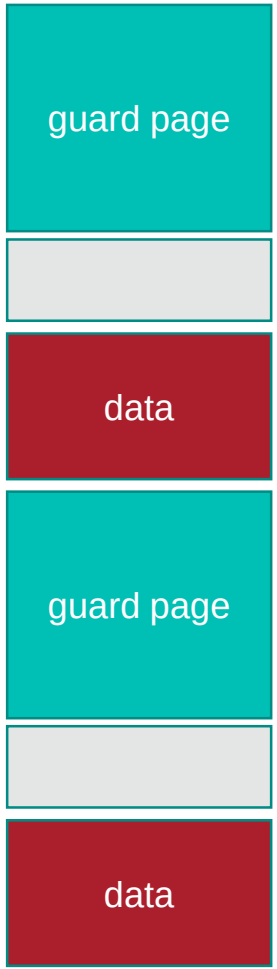
# Electric Fence

- Use guard page
  - Page with memory protection so that if touched, create a fault
- Fault immediate
- No extra code check
- What may be the problem?



# Electric Fence

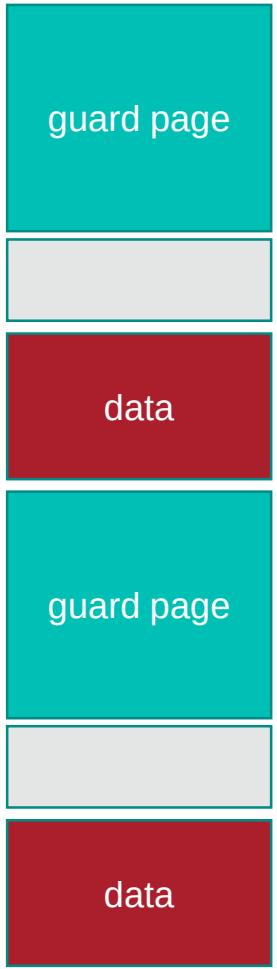
- Use guard page
  - Page with memory protection so that if touched, create a fault
- Fault immediate
- No extra code check
- Very memory inefficient
- Work only across pages
- Generally used only for debugging/test



# Electric Fence

Homework/exam question: Describe the page guard technique to protect from buffer overflow exploits.

- Use guard page
  - Page with memory protection so that if touched, create a fault
- Fault immediate
- No extra code check
- Very memory inefficient
- Work only across pages
- Generally used only for debugging/test



# Bounds checking

- Make sure pointer refer to a specific memory object, and does not go out of that object

# Bounds checking

- Make sure pointer refer to a specific memory object, and does not go out of that object
  - Check can be added automatically at compilation time…

# Bounds checking

- Make sure pointer refer to a specific memory object, and does not go out of that object
- Easy on paper…
- … a bit harder in C

# Bounds checking

- Make sure pointer refer to a specific memory object, and does not go out of that object
- Easy on paper…
- … a bit harder in C
  - char x[1024];
  - char *y = x[107];

# Bounds checking

- Make sure pointer refer to a specific memory object, and does not go out of that object
- Easy on paper…
- … a bit harder in C
  - char x[1024];
  - char *y = x[107];

  - union{
  -  int c;
  -  struct s{
  -  int j;
  -  int k;
  - }};

# Bounds checking

- Make sure pointer refer to a specific memory object, and does not go out of that object
- Easy on paper…
- … a bit harder in C
  - char x[1024];
  - char *y = x[107];

  - union{
  -  int c;
  -  struct s{
  -  int j;
  -  int k;
  - }};

- `int *ptr = &(p);`

# Bounds checking

- Make sure pointer refer to a specific memory object, and does not go out of that object
- Easy on paper…
- … a bit harder in C
  - char x[1024];
  - char *y = x[107];

  - union{
  - int c;
  - struct s{
  - int j;
  - int k;
  - }};

- `int *ptr = &(p);`

- Weaker guarantees
  - From a pointer p' deriving from p. Then p' should only be used to dereference memory that belongs to p.

# Bounds checking

- Make sure pointer refer to a specific memory object, and does not go out of that object
- Easy on paper…
- … a bit harder in C
  - char x[1024];
  - char *y = x[107];

  - union{
  - int c;
  - struct s{
  - int j;
  - int k;
  - }};

- `int *ptr = &(p);`

- Weaker guarantees
  - From a pointer p' deriving from p. Then p' should only be used to dereference memory that belongs to p.
- You can still trample memory
- … but not arbitrary memory

# Bounds checking

- Make sure pointer refer to a specific memory object, and does not go out of that object
- Easy on paper…
- … a bit harder in C
  - char x[1024];
  - char *y = x[107];
  - union{
  - int c;
  - struct s{
  - int j;
  - int k;
  - }};

- `int *ptr = &(p);`
- Weaker guarantees
  - From a pointer p' deriving from p. Then p' should only be used to dereference memory that belongs to p.
- You can still trample memory
- … but not arbitrary memory

Requires compiler support: issue with legacy libraries

# Bounds checking

Homework/exam question:
Describe bound checking techniques to protect from buffer overflow vulnerabilities.

- Make sure pointer refer to a specific memory object, and does not go out of that object
- Easy on paper…
- … a bit harder in C
  - char x[1024];
  - char *y = x[107];

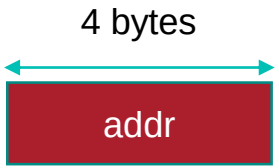
  - union{
  - int c;
  - struct s{
  - int j;
  - int k;
  - }};

- `int *ptr = &(p);`

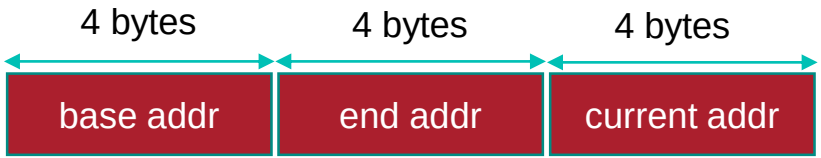
- Weaker guarantees
  - From a pointer p' deriving from p. Then p' should only be used to dereference memory that belongs to p.
- You can still trample memory
- … but not arbitrary memory

# Fat address

- 32 bits address

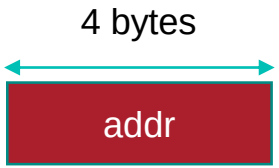



- fat address



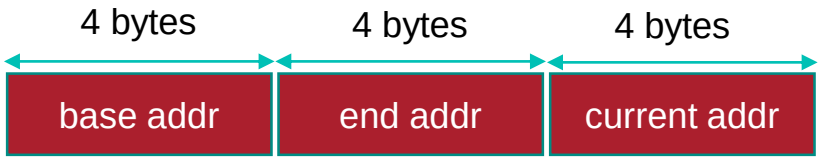
# Fat address

- 32 bits address



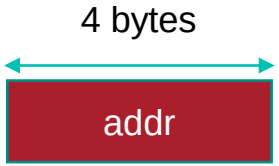

```
int *ptr = malloc(8);
While(1) {
   *ptr = 42;
   ptr++;
}
```

- fat address



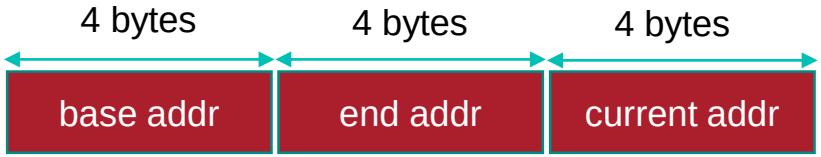
# Fat address

- 32 bits address




```
int *ptr = malloc(8);
While(1) {
  *ptr = 42;
  ptr++;
}
```

- fat address



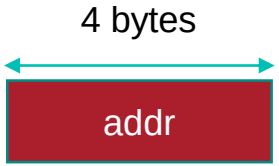

Need to instrument code
i.e. compiler support

Problem with external library
Non-atomic

# Fat address

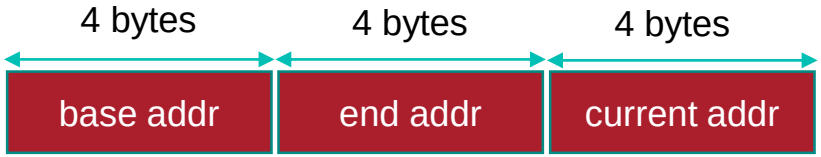
Homework/exam question:
Describe fat addresses techniques to protect from buffer overflow vulnerabilities.

- 32 bits address




```
int *ptr = malloc(8);
While(1) {
  *ptr = 42;
  ptr++;
}
```

- fat address




Need to instrument code
i.e. compiler support

Problem with external library
Non-atomic

University of BRISTOL

# Worms

… and a bit of history

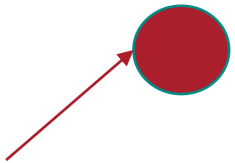
bristol.ac.uk

# Morris Worm 1988

- a.k.a the Great Worm
- Designed by Robert Morris in 1988

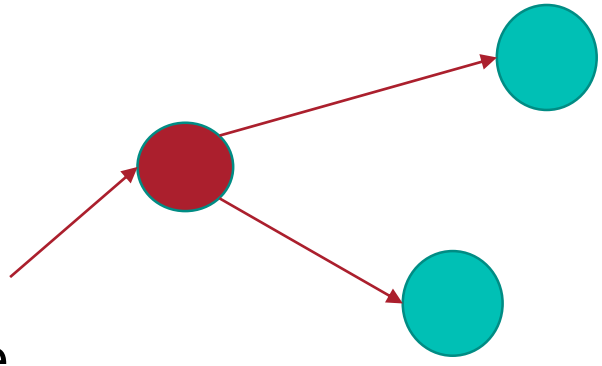
# Morris Worm 1988

- a.k.a the Great Worm
- Designed by Robert Morris
- Exploit a vulnerability to execute a program on a machine

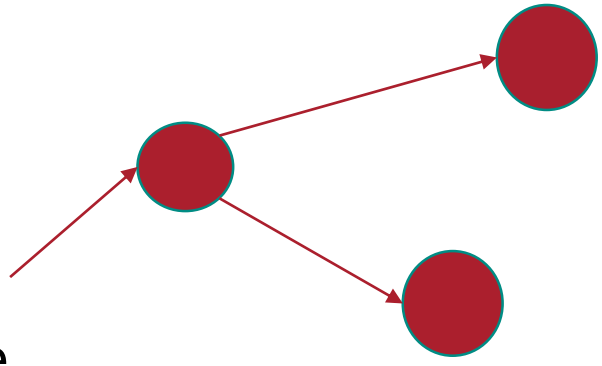


# Morris Worm 1988

- a.k.a the Great Worm
- Designed by Robert Morris
- Exploit a vulnerability to execute a program on a machine
- Send payload to compromise other machines



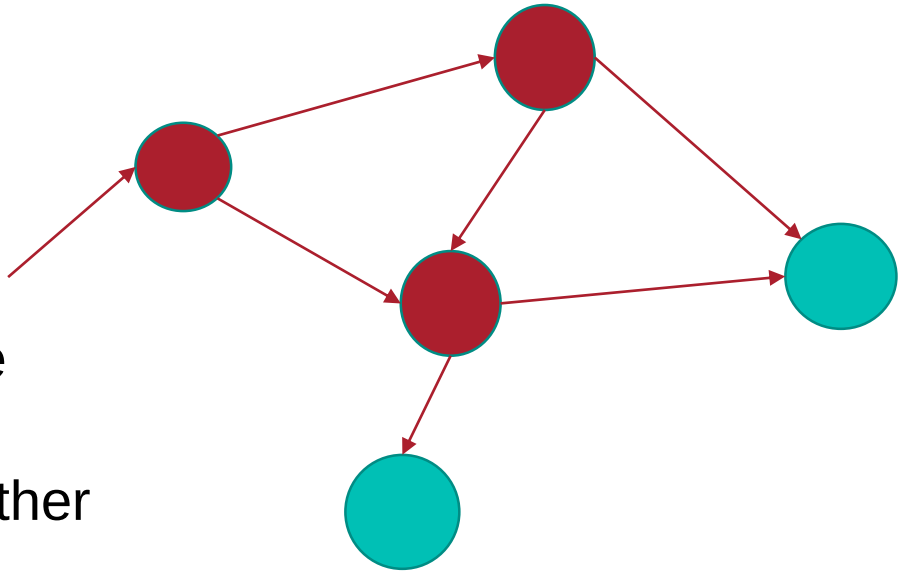
# Morris Worm 1988



- a.k.a the Great Worm
- Designed by Robert Morris
- Exploit a vulnerability to execute a program on a machine
- Send pay load to compromise other machines on the same network

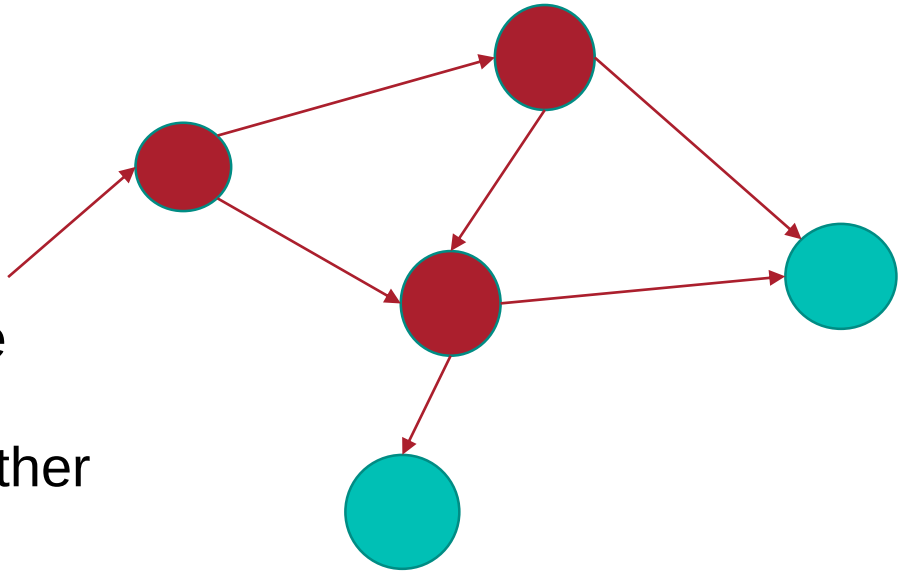
# Morris Worm 1988

- a.k.a the Great Worm
- Designed by Robert Morris
- Exploit a vulnerability to execute a program on a machine
- Send pay load to compromise other machines on the same network
- Repeat



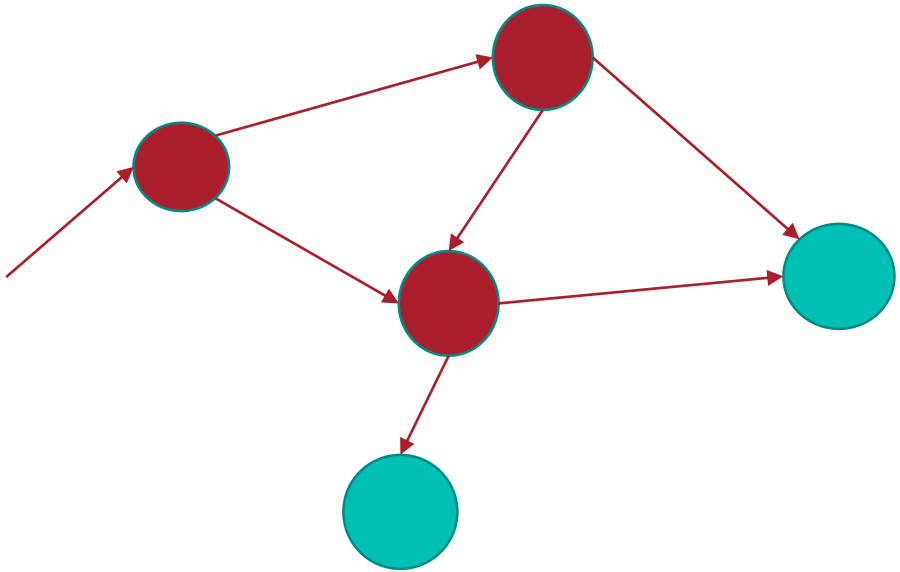
# Morris Worm 1988

- a.k.a the Great Worm
- Designed by Robert Morris
- Exploit a vulnerability to execute a program on a machine
- Send pay load to compromise other machines on the same network
- Repeat
- Stated purpose “mapping” the internet



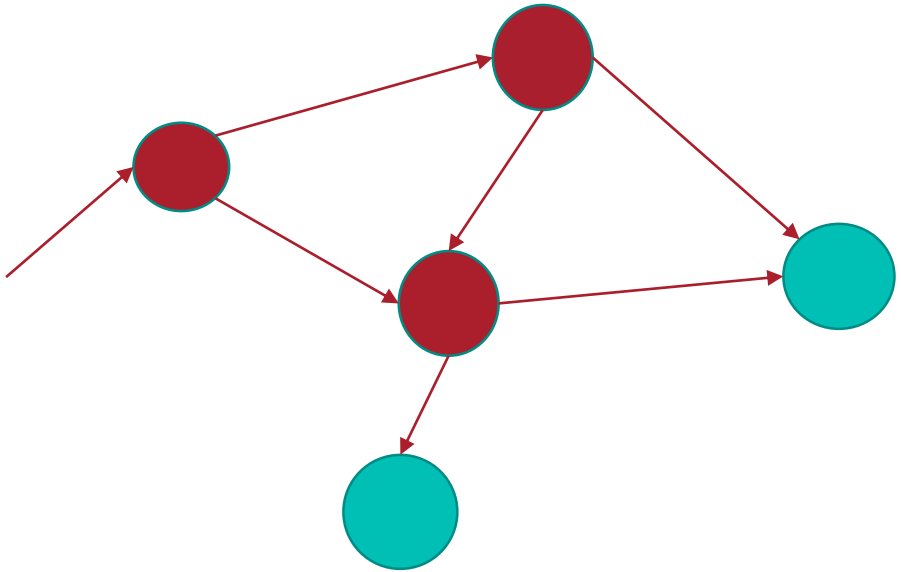
# Morris Worm 1988

- Worm could test if a copy was already there by asking



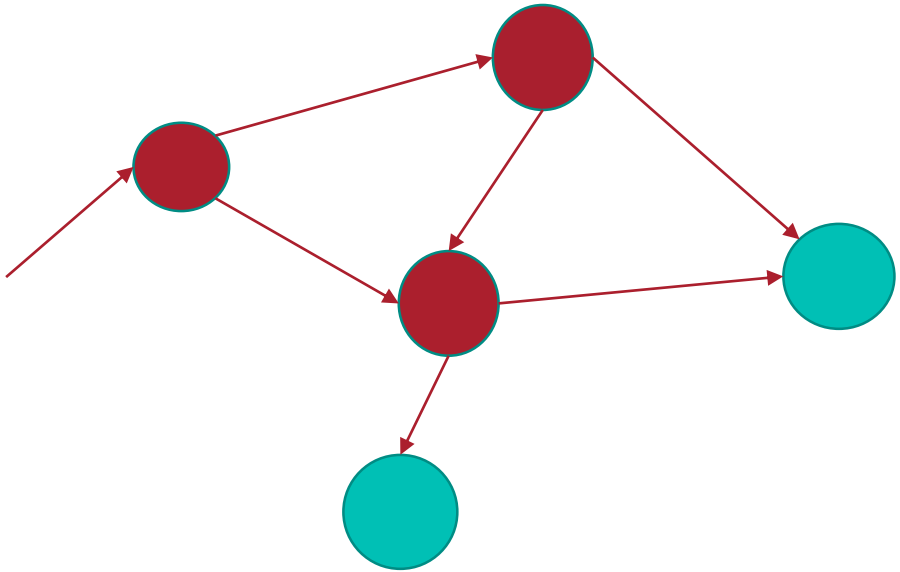
# Morris Worm 1988

- Worm could test if a copy was already there by asking
  - Countermeasure?



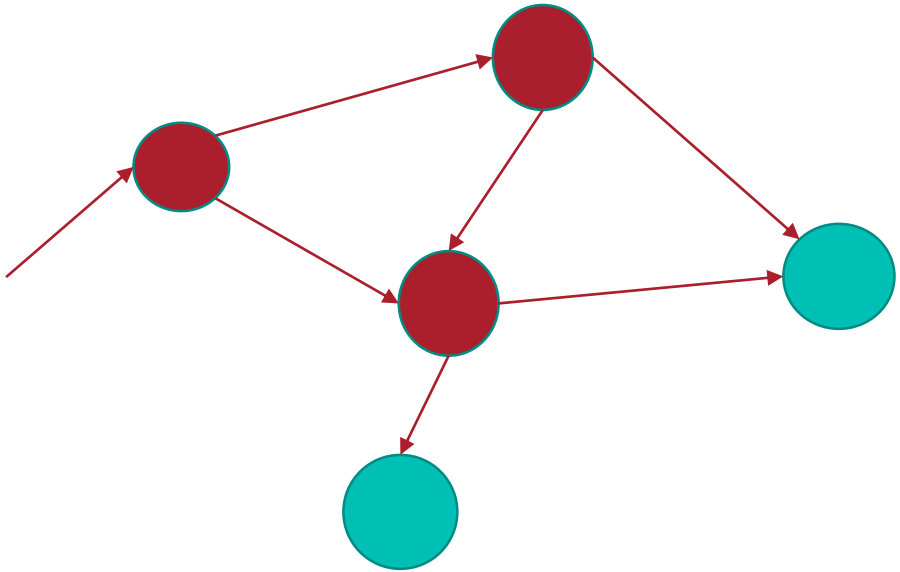
# Morris Worm 1988

- Worm could test if a copy was already there by asking
  - Countermeasure?
  - Simply say yes



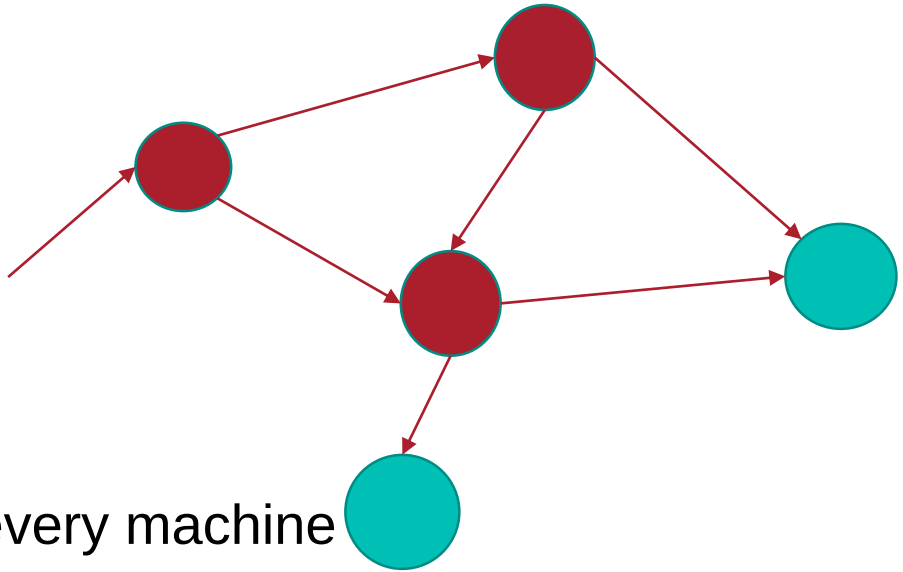
# Morris Worm 1988

- Worm could test if a copy was already there by asking
  - Countermeasure?
  - Simply say yes
  - Copy anyway 1/7 time



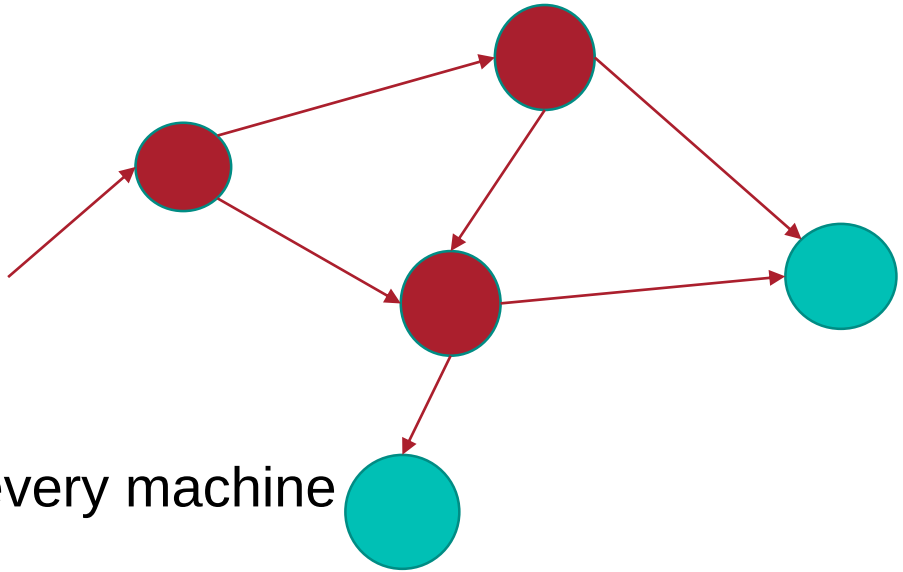
# Morris Worm 1988

- Worm could test if a copy was already there by asking
  - Countermeasure?
  - Simply say yes
  - Copy anyway 1/7 time
- Results thousands process on every machine
- Machine running to a crawl

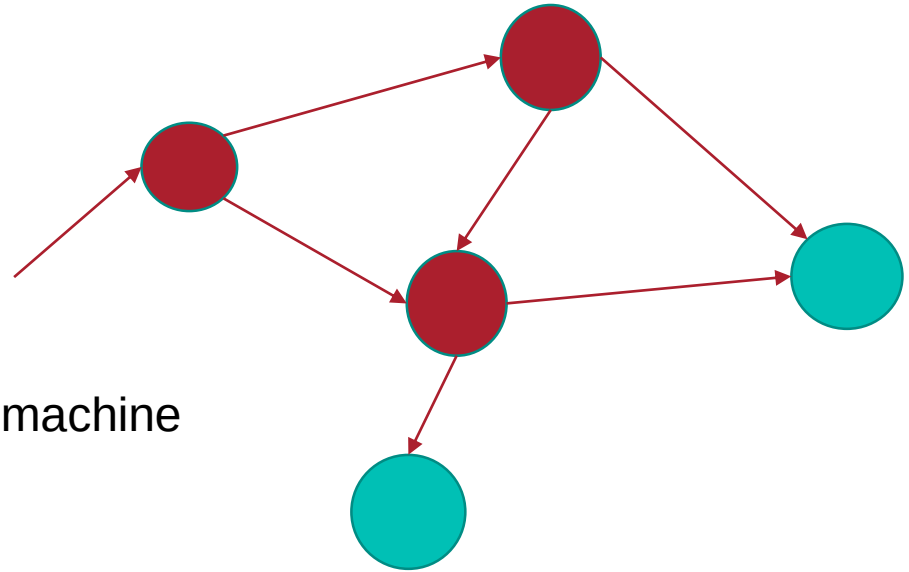


# Morris Worm 1988

- Worm could test if a copy was already there by asking
  - Countermeasure?
  - Simply say yes
  - Copy anyway 1/7 time
- Results thousands process on every machine
- Machine running to a crawl
- Take down a machine to clean it
- Get infected again instantly



# Morris Worm 1988



- Worm could test if a copy was already there by asking
  - Countermeasure?
  - Simply say yes
  - Copy anyway 1/7 time
- Results thousands process on every machine
- Machine running to a crawl
- Take down a machine to clean it
- Get reinfected instantly
- Required a coordinated effort to “clean” the internet
- Largest denial of service attack

# Blaster

- 2003 – Affect Windows 2000/XP Machines
- Exploit buffer overflow vulnerability on Remote Procedure Call
  - Get a shell with “admin” privilege
  - To download payload via ftp
  - And install it

# Blaster

- 2003 – Affect Windows 2000/XP Machines
- Exploit buffer overflow vulnerability on Remote Procedure Call
- Aim to remain undetected
  - No more thousands processes
  - Check existence of a mutex (“BILLY”)
- Infect other random machine on the network
- Variant A – start a thread to DDOS Microsoft update

# Blaster

- 2003 – Affect Windows 2000/XP Machines
- Exploit buffer overflow vulnerability on Remote Procedure Call
- Aim to remain undetected
- Infect other random machine on the network
- Variant A – start a thread to DDOS Microsoft update
- Contains two messages
  - I just want to say LOVE YOU SAN!!
  - billy gates why do you make this possible ? Stop making money and fix your software!!

# Blaster

- 2003 – Affect Windows 2000/XP Machines
- Exploit buffer overflow vulnerability on Remote Procedure Call
- Aim to remain undetected
- Infect other random machine on the network
- Variant A – start a thread to DDOS Microsoft update
- Later variant caused system to reboot every 60 seconds

# Other buffer overflow example

- Twilight Hack (Wii)
  - Buffer Overflow on Legend of Zelda: Twilight Princess
  - When reading save files
  - Used to install pirated games

# Buffer overflow in 2019? (just one of many)

## CVE-2019-10164 Detail

MODIFIED

This vulnerability has been modified since it was last analyzed by the NVD. It is awaiting reanalysis which may result in further changes to the information provided.

## Current Description

PostgreSQL versions 10.x before 10.9 and versions 11.x before 11.4 are vulnerable to a stack-based buffer overflow. Any authenticated user can overflow a stack-based buffer by changing the user's own password to a purpose-crafted value. This often suffices to execute arbitrary code as the PostgreSQL operating system account.

# Buffer overflow in 2019? (just one of many)

Homework/exam question:
Buffer overflow attacks have been known for decade. Yet they occur. Discuss.

## CVE-2019-10164 Detail

MODIFIED

This vulnerability has been modified since it was last analyzed by the NVD. It is awaiting reanalysis which may result in further changes to the information provided.

## Current Description

PostgreSQL versions 10.x before 10.9 and versions 11.x before 11.4 are vulnerable to a stack-based buffer overflow. Any authenticated user can overflow a stack-based buffer by changing the user's own password to a purpose-crafted value. This often suffices to execute arbitrary code as the PostgreSQL operating system account.

University of BRISTOL

# Thank you

Office MVB 3.26

bristol.ac.uk