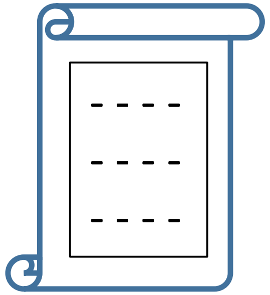
Buffer Overflow Lab

COSC 458 – 647

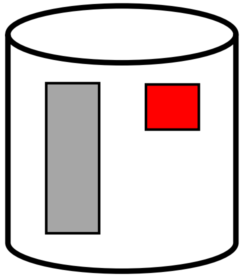
Towson University

Overview

badfile



./stack



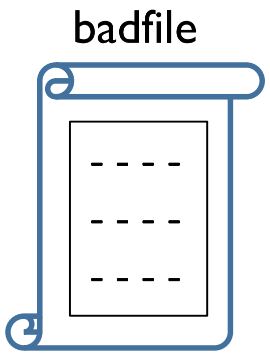
1. ./stack is a precompiled program that has two string buffers `str_main[517]` in main(), and `buff[24]` in bof() methods.
2. It main task is to open and read data from a file named `badfile`.
3. ./stack then copies the read data to its own string buffer `str_main`,

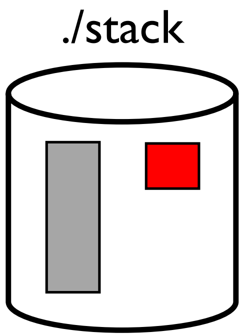
   and then, to its smaller string buffer `buff`.

   1. This creates a chance for buffer overflow (How ?)
4. *If `badfile` contains malicious data/code, this BOF can trigger ./stack to intentionally execute that code.*

This is what we will explore in this lab.

Overview





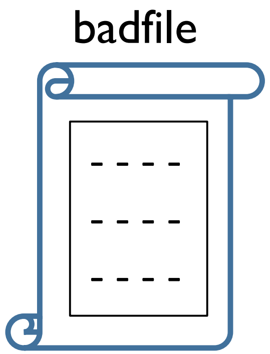
1. ./stack is a precompiled program that has two string buffers `str_main[517]` in main(), and `buff[24]` in bof() methods.
2. It main task is to open and read data from a file named `badfile`.
3. ./stack then copies the read data to its own string buffer `str_main`,

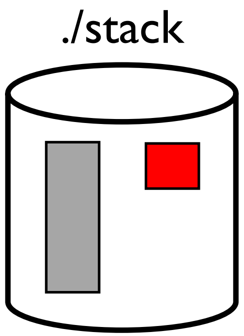
   and then, to its smaller string buffer `buff`.

   1. This creates a chance for buffer overflow (How ?)
4. *If `badfile` contains malicious data/code, this BOF can trigger ./stack to intentionally execute that code.*

This is what we will explore in this lab.

Overview

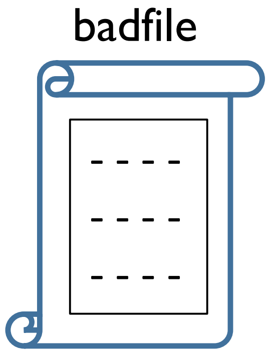


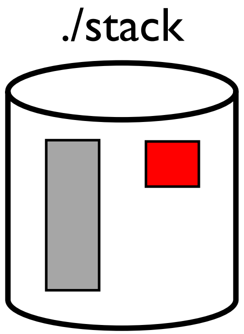


1. ./stack is a precompiled program that has two string buffers `str_main[517]` in main(), and `buff[24]` in bof() methods.
2. It main task is to open and read data from a file named `badfile`.
3. ./stack then copies the read data to its own string buffer `str_main`, and then, to its smaller string buffer `buff`.
    1. This creates a chance for buffer overflow (How ?)
4. *If `badfile` contains malicious data/code, this BOF can trigger ./stack to intentionally execute that code.*

This is what we will explore in this lab.

Overview

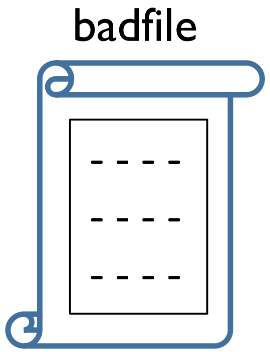


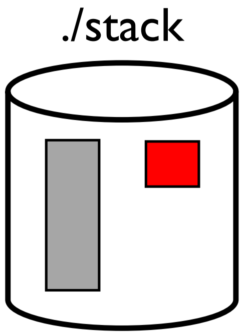


1. ./stack is a precompiled program that has two string buffers **`str_main[517]`** in main(), and **`buff[24]`** in bof() methods.
2. It main task is to open and read data from a file named **`badfile`**.
3. ./stack then copies the read data to its own string buffer **`str_main`**, and then, to its smaller string buffer **`buff`**.
    1. This creates a chance for buffer overflow (How ?)
4. *If **`badfile`** contains malicious data/code, this BOF can trigger ./stack to intentionally execute that code.*

This is what we will explore in this lab.

Overview





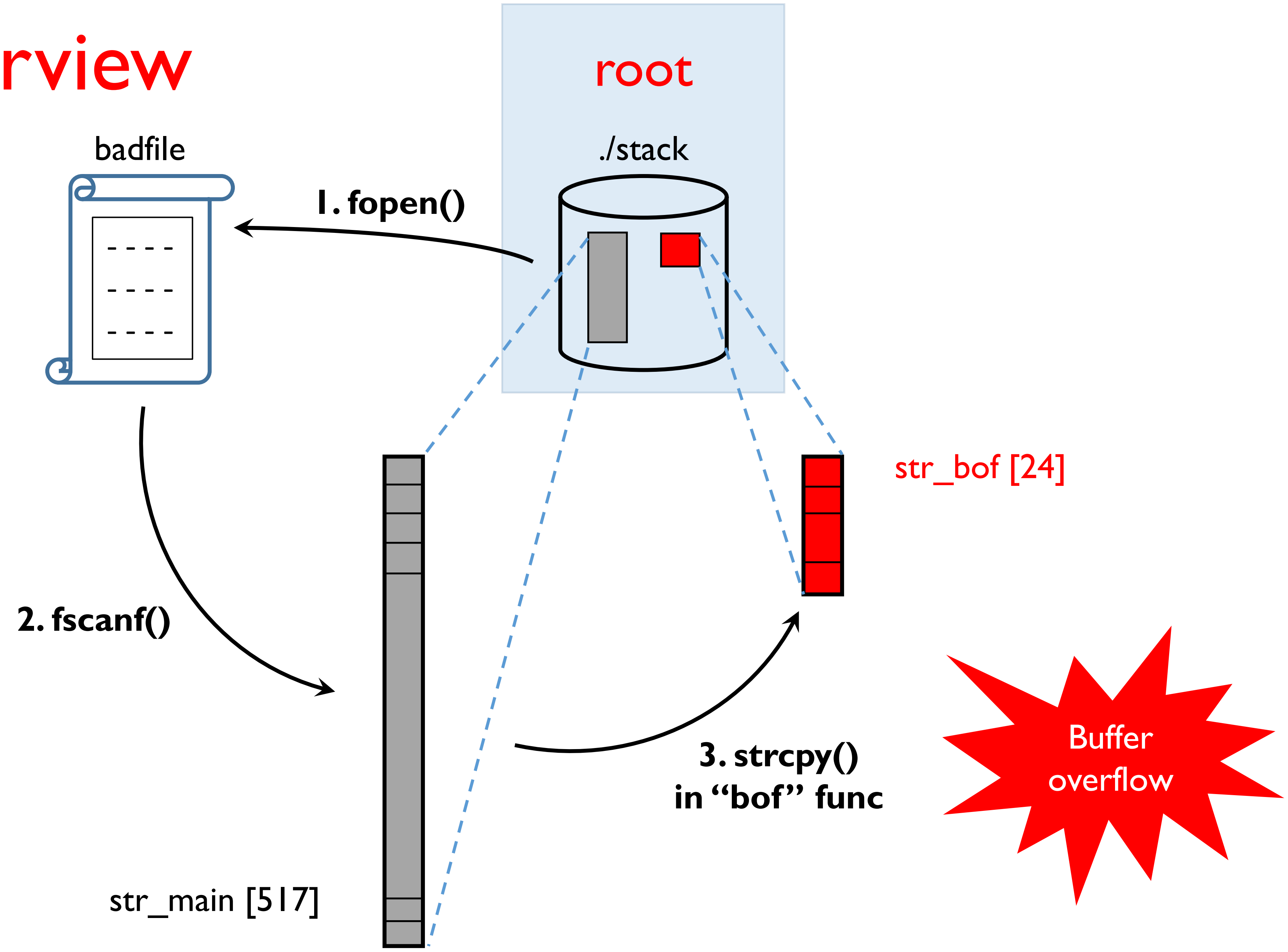
1. ./stack is a precompiled program that has two string buffers **`str_main[517]`** in main(), and **`buff[24]`** in bof() methods.
2. It main task is to open and read data from a file named **`badfile`**.
3. ./stack then copies the read data to its own string buffer **`str_main`**,

   and then, to its smaller string buffer **`buff`**.

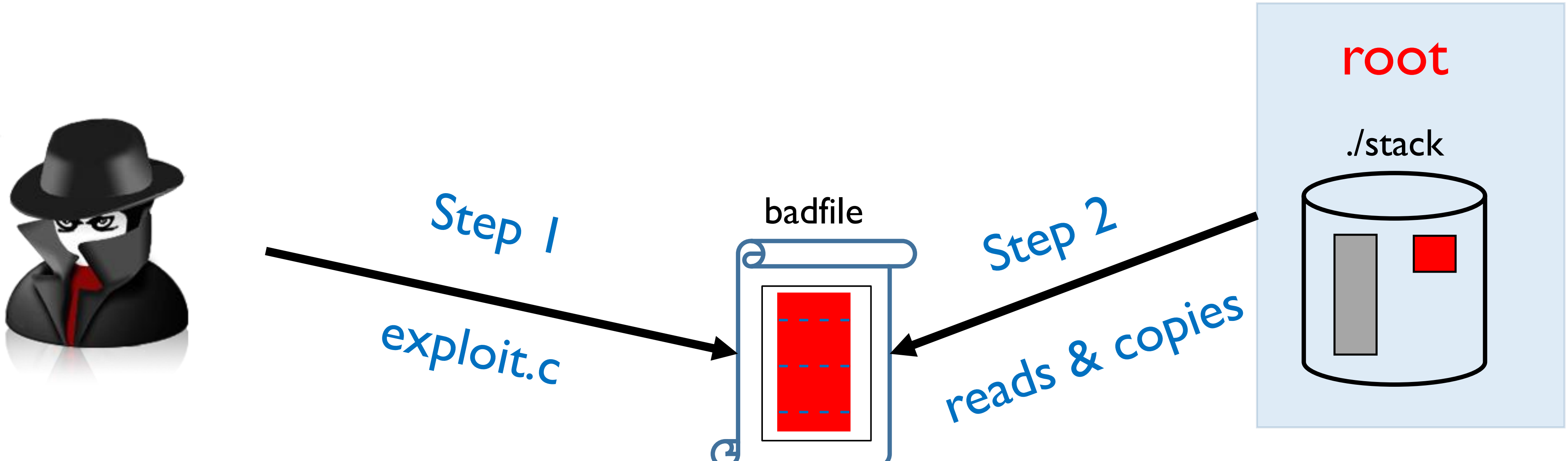
    1. This creates a chance for buffer overflow (How ?)
4. *If **`badfile`** contains malicious data/code, this BOF can trigger ./stack to intentionally execute that code.*

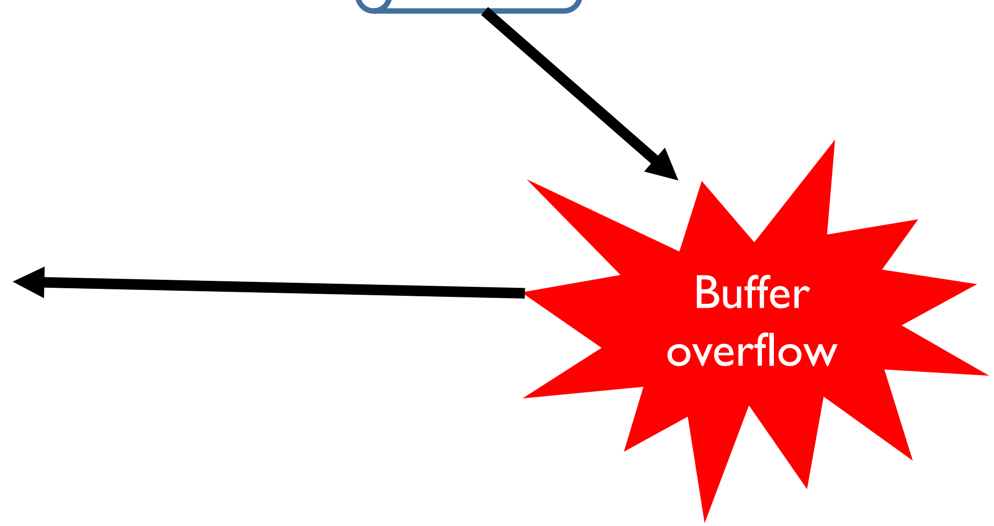
This is what we will explore in this lab.

Overview


root
./stack
badfile
1. fopen()
2. fscanf()
3. strcpy()
in "bof" func
str_bof [24]
str_main [517]
Buffer
overflow

Overview





```
struct group_info init_groups = { .usage = ATOMIC_INIT(2) };
struct group_info *groups_alloc(int gidsetsize){
    struct group_info *group_info;
    int nblocks;
    int i;

    nblocks = (gidsetsize + NGROUPS_PER_BLOCK - 1) / NGROUPS_PER_BLOCK;
    /* Make sure we always allocate at least one indirect block pointer */
    nblocks = nblocks ? : 1;
    group_info = kmalloc(sizeof(*group_info|
```

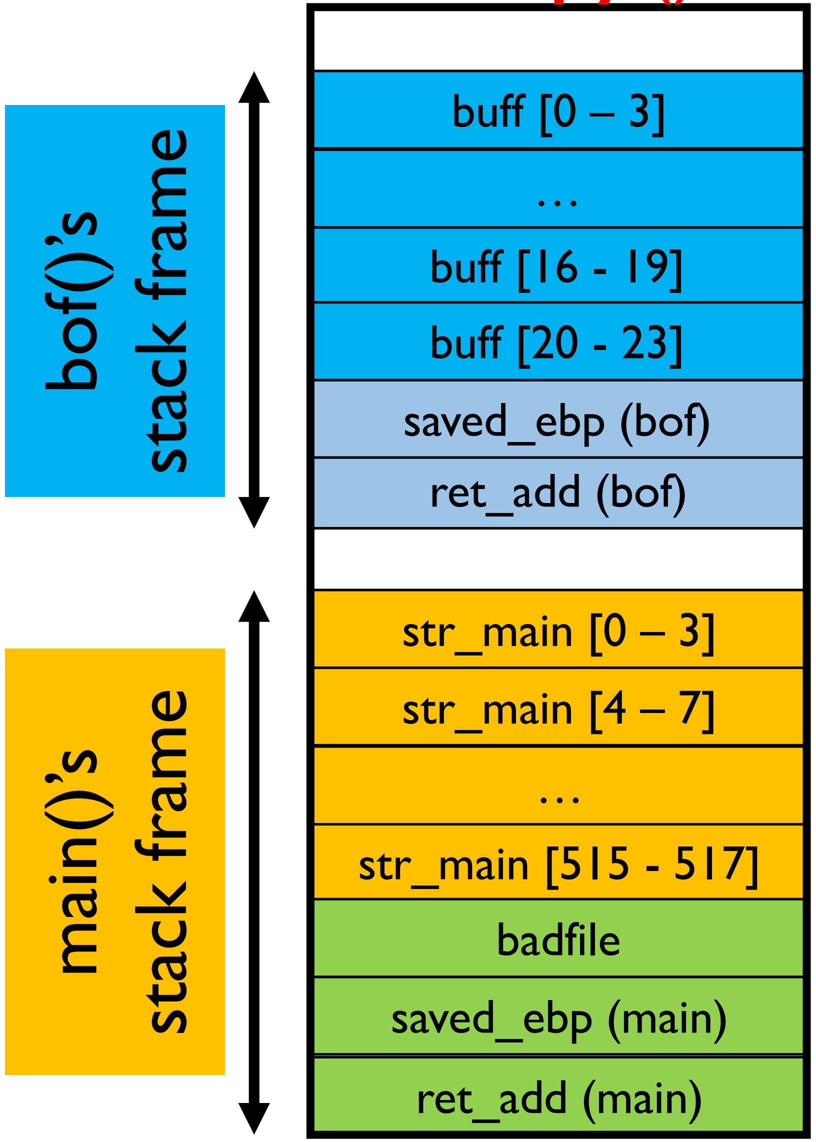
ACCESS GRANTED

BOF - Explained

```c
/* stack.c */
int bof(char *str)  {
  char buffer[24];
  strcpy(buffer, str);
  return 1;
}

int main(int argc, char **argv) {
  char str[517];
  FILE *badfile;

  badfile = fopen("badfile", "r");
  fread(str, 1, 517, badfile);
  bof(str);

  printf("Returned Properly\n");
  return 1;
}
```

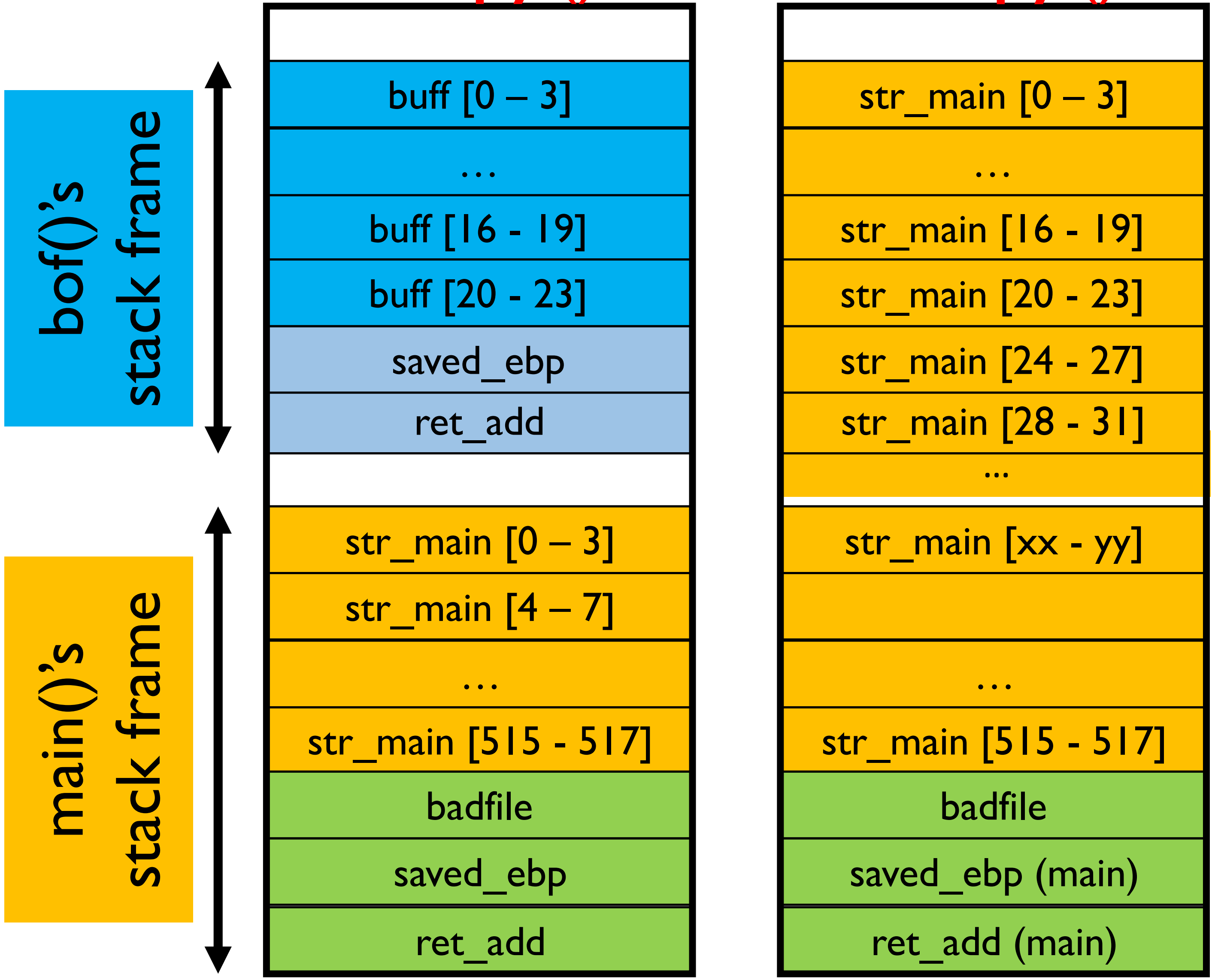
BOF - Explained

Before strcpy()



```
/* stack.c */
int bof(char *str)  {
   char buffer[24];
   strcpy(buffer, str);
   return 1;
}

int main(int argc, char **argv) {
   char str[517];
   FILE *badfile;

   badfile = fopen("badfile", "r");
   fread(str, 1, 517, badfile);
   bof(str);

   printf("Returned Properly\n");
   return 1;
}
```

BOF - Explained



Before strcpy()

After strcpy()

bof()'s stack frame

main()'s stack frame

Before strcpy()	After strcpy()
buff [0 – 3]	str_main [0 – 3]
…	…
buff [16 - 19]	str_main [16 - 19]
buff [20 - 23]	str_main [20 - 23]
saved_ebp	str_main [24 - 27]
ret_add	str_main [28 - 31]
	…
str_main [0 – 3]	str_main [xx - yy]
str_main [4 – 7]	
…	…
str_main [515 - 517]	str_main [515 - 517]
badfile	badfile
saved_ebp	saved_ebp (main)
ret_add	ret_add (main)

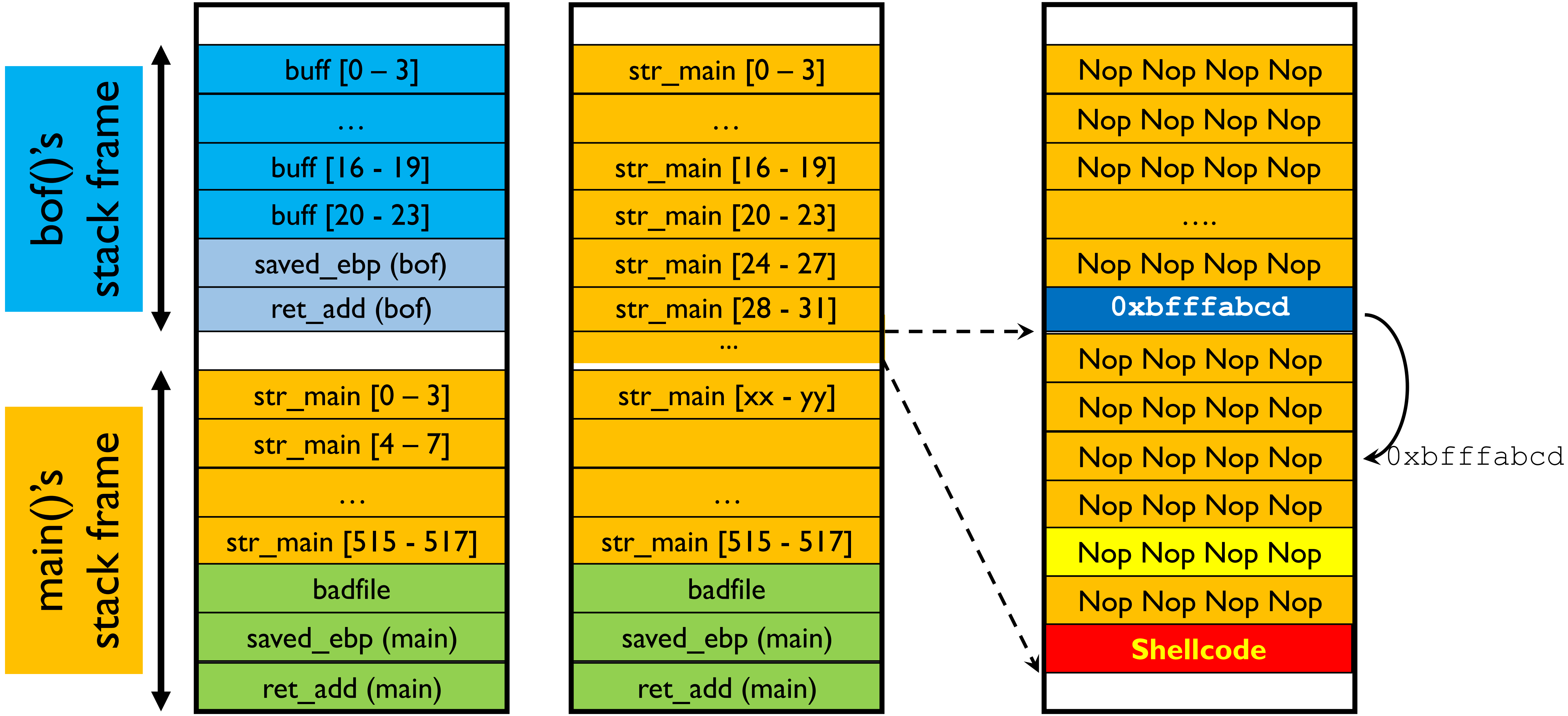
```
/* stack.c */
int bof(char *str)  {
   char buffer[24];
   strcpy(buffer, str);
   return 1;
}

int main(int argc, char **argv) {
   char str[517];
   FILE *badfile;

   badfile = fopen("badfile", "r");
   fread(str, 1, 517, badfile);
   bof(str);

   printf("Returned Properly\n");
   return 1;
}
```

What should badfile contain?

- Bad data/code – Of course

- Can it be both data and code?
  - If yes, do we want the code to be executed automatically?
  - How do we do that?

- What should it actually contain?



bof()'s stack frame

main()'s stack frame

Before

- buff [0 – 3]
- …
- buff [16 - 19]
- buff [20 - 23]
- saved_ebp (bof)
- ret_add (bof)
- str_main [0 – 3]
- str_main [4 – 7]
- …
- str_main [515 - 517]
- badfile
- saved_ebp (main)
- ret_add (main)

After

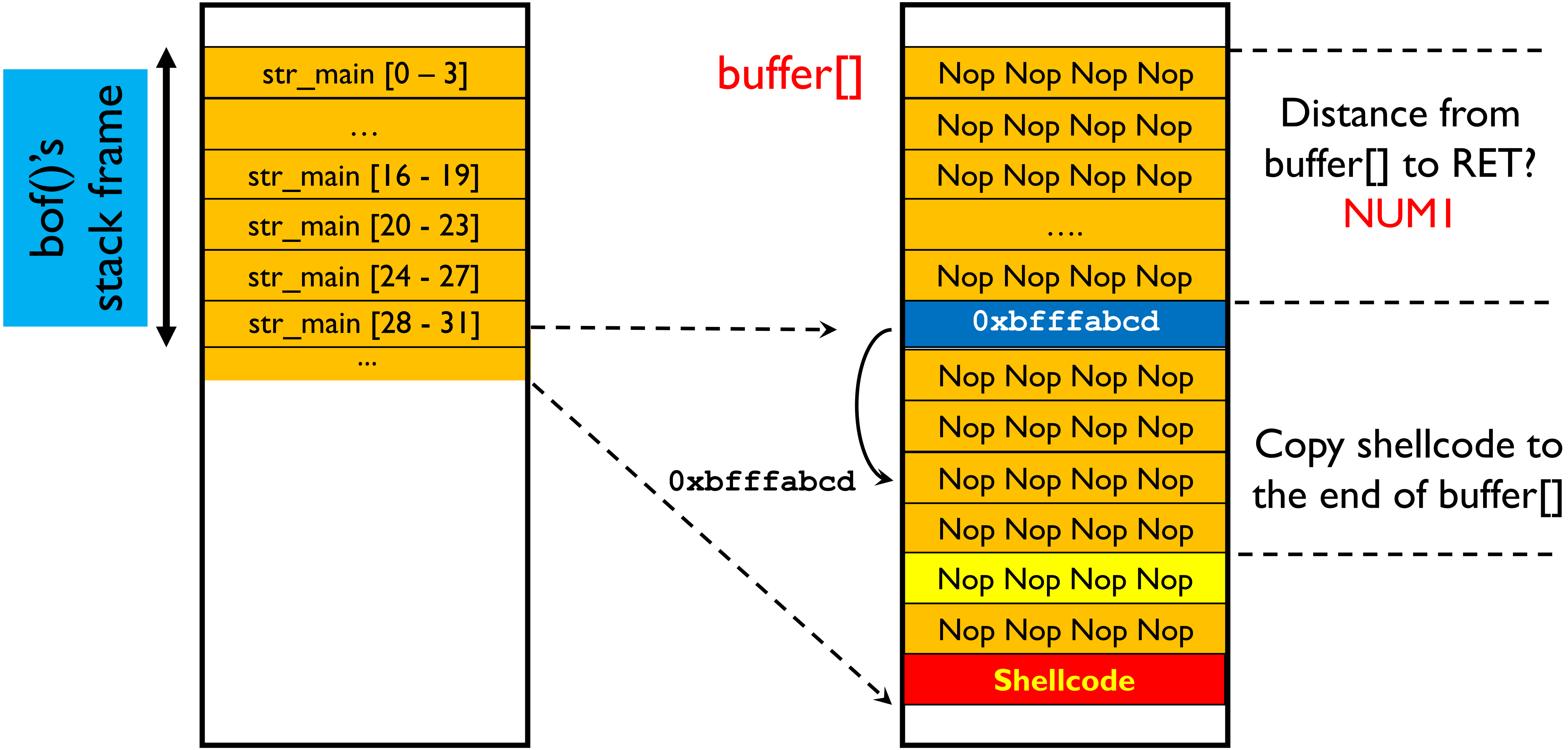
- str_main [0 – 3]
- …
- str_main [16 - 19]
- str_main [20 - 23]
- str_main [24 - 27]
- str_main [28 - 31]
- …
- str_main [xx - yy]
- …
- str_main [515 - 517]
- badfile
- saved_ebp (main)
- ret_add (main)

Attack vector

- Nop Nop Nop Nop
- Nop Nop Nop Nop
- Nop Nop Nop Nop
- ….
- Nop Nop Nop Nop
- 0xbfffabcd
- Nop Nop Nop Nop
- Nop Nop Nop Nop
- Nop Nop Nop Nop
- Nop Nop Nop Nop
- Nop Nop Nop Nop
- Nop Nop Nop Nop
- Shellcode

After

exploit.c



Distance from buffer[] to RET?
NUM1

Copy shellcode to the end of buffer[]

Steps

1. Debug stack in `gdb`, find the addresses of `buffer[]` and `ebp` in `bof()`

2. Estimate the distance between `buffer[]` and the return address

3. In **`exploit.c`**
    1. pick a return address of your choice and copy it in the right place in the buffer.
    2. Copy the shellcode to the buffer once you have set the attack return address

Attack vector - 3 things

1. The address
    - “0xa1b2c3d4” will not work – it is just an example
    - This can be found by debugging the relative address of “**ret_add (bof)**”.

2. NUM1: Try multiple numbers greater than 24.

3. NUM2: Try multiple numbers greater than 20.