Hacking in C

Attacks, part III

Radboud University, Nijmegen, The Netherlands

Spring 2018

A short recap

- Overwrite the return address: Manipulating program flow
  - Return to other **existing code**
  - Return to code **that we inject** $\leftarrow$ last week

A short recap

- Overwrite the return address: Manipulating program flow
  - Return to other **existing code**
  - Return to code **that we inject** $\leftarrow$ last week
- Shell code: tiny piece of assembly code that spawns a shell
  - Stay clear of NULL-bytes

A short recap

- Overwrite the return address: Manipulating program flow
  - Return to other **existing code**
  - Return to code **that we inject** ← last week
- Shell code: tiny piece of assembly code that spawns a shell
  - Stay clear of NULL-bytes
- Mitigations

A short recap

- Overwrite the return address: Manipulating program flow
  - Return to other **existing code**
  - Return to code **that we inject** ← last week
- Shell code: tiny piece of assembly code that spawns a shell
  - Stay clear of NULL-bytes
- Mitigations
  - Less code means less liability

A short recap

- Overwrite the return address: Manipulating program flow
  - Return to other **existing code**
  - Return to code **that we inject** $\leftarrow$ last week
- Shell code: tiny piece of assembly code that spawns a shell
  - Stay clear of NULL-bytes
- Mitigations
  - Less code means less liability
  - `libsafe`

A short recap

- Overwrite the return address: Manipulating program flow
  - Return to other **existing code**
  - Return to code **that we inject** ← last week
- Shell code: tiny piece of assembly code that spawns a shell
  - Stay clear of NULL-bytes
- Mitigations
  - Less code means less liability
  - `libsafe`
  - Dynamic analysis (`valgrind`, `clang`'s AddressSanitizer)

A short recap

- Overwrite the return address: Manipulating program flow
  - Return to other **existing code**
  - Return to code **that we inject** ← last week
- Shell code: tiny piece of assembly code that spawns a shell
  - Stay clear of NULL-bytes
- Mitigations
  - Less code means less liability
  - `libsafe`
  - Dynamic analysis (`valgrind`, `clang`'s AddressSanitizer)
  - Static analysis (CCured, PREfast, Flawfinder)

A short recap

- Overwrite the return address: Manipulating program flow
  - Return to other **existing code**
  - Return to code **that we inject** ← last week
- Shell code: tiny piece of assembly code that spawns a shell
  - Stay clear of NULL-bytes
- Mitigations
  - Less code means less liability
  - `libsafe`
  - Dynamic analysis (`valgrind`, `clang`'s AddressSanitizer)
  - Static analysis (CCured, PREfast, Flawfinder)
  - Stack canaries

A short recap

- Overwrite the return address: Manipulating program flow
  - Return to other **existing code**
  - Return to code **that we inject** ← last week
- Shell code: tiny piece of assembly code that spawns a shell
  - Stay clear of NULL-bytes
- Mitigations
  - Less code means less liability
  - `libsafe`
  - Dynamic analysis (`valgrind`, `clang`'s AddressSanitizer)
  - Static analysis (CCured, PREfast, Flawfinder)
  - Stack canaries
  - Data Execution Prevention

Return to libc

- Attacker cannot execute his code on the stack anymore
- Workaround: execute code that is already in the program
- (Almost) always mapped into the programs memory space: C standard library

Return to libc

- Attacker cannot execute his code on the stack anymore
- Workaround: execute code that is already in the program
- (Almost) always mapped into the programs memory space: C standard library
- Idea:
  - Somehow prepare arguments for `system()`
  - overwrite return address with address of `system()`

Return to libc

- Attacker cannot execute his code on the stack anymore
- Workaround: execute code that is already in the program
- (Almost) always mapped into the programs memory space: C standard library
- Idea:
  - Somehow prepare arguments for `system()`
  - overwrite return address with address of `system()`
- Obtain the address of `libc` with
  `cat /proc/$PID/maps | grep libc`
- Obtain the offset of `system()` through

```
nm -D /lib/x86_64-linux-gnu/libc.so.6 | grep system
```

Preparing arguments

```
int system(const char *command);
```

- Target: first argument to `system()` should be address of `"/bin/sh"`
- Can write `"/bin/sh"` somewhere
- Alternative: find `"/bin/sh"` somewhere in the binary or libraries
- Then obtain address of `"/bin/sh"`

"The old days" (x86)

- Arguments are passed through the stack
- Write behind buffer
  1. Address of `system()`
  2. Address of `exit()`
  3. Address of `"/bin/sh"`

"The old days" (x86)

- Arguments are passed through the stack
- Write behind buffer
  1. Address of `system()`
  2. Address of `exit()`
  3. Address of `"/bin/sh"`
- Address of `system()` must overwrite return address in current frame
- Code will return to `system()` with
  - return address pointing to `exit()`, and
  - argument pointing to `"/bin/sh"`

Nowadays: AMD64 (x64, x86-64)

- Arguments are passed through registers
- Somehow need to get the address of `"/bin/sh"` into `%rdi`

Nowadays: AMD64 (x64, x86-64)

- Arguments are passed through registers
- Somehow need to get the address of `"/bin/sh"` into `%rdi`
- Idea: find "gadget"
  ```
  pop %rdi
  retq
  ```
- Overwrite return address with that gadget

Nowadays: AMD64 (x64, x86-64)

- Arguments are passed through registers
- Somehow need to get the address of `"/bin/sh"` into `%rdi`
- Idea: find "gadget"
  ```
  pop %rdi
  retq
  ```
- Overwrite return address with that gadget
- Put address of `"/bin/sh"` behind this new "return address"
- Put address of `system()` behind

Nowadays: AMD64 (x64, x86-64)

- Arguments are passed through registers
- Somehow need to get the address of `"/bin/sh"` into `%rdi`
- Idea: find "gadget"
  ```
  pop %rdi
  retq
  ```
- Overwrite return address with that gadget
- Put address of `"/bin/sh"` behind this new "return address"
- Put address of `system()` behind
- What will happen?:
  - Gadget will pop the address of `"/bin/sh"` into `%rdi`
  - `retq` will return into `system()`

Nowadays: AMD64 (x64, x86-64)

- Arguments are passed through registers
- Somehow need to get the address of `"/bin/sh"` into `%rdi`
- Idea: find "gadget"
  ```
  pop %rdi
  retq
  ```
- Overwrite return address with that gadget
- Put address of `"/bin/sh"` behind this new "return address"
- Put address of `system()` behind
- What will happen?:
  - Gadget will pop the address of `"/bin/sh"` into `%rdi`
  - `retq` will return into `system()`
- ROP-technique generalizes this (later)

Countermeasures

- Can make sure that `\0` is in the address of libc
- Many functions (like `gets`) won't read past the `\0`

Countermeasures

- Can make sure that `\0` is in the address of libc
- Many functions (like `gets`) won't read past the `\0`
- Does not generally help, can overflow some buffers also with `\0`

Countermeasures

- Can make sure that `\0` is in the address of libc
- Many functions (like `gets`) won't read past the `\0`
- Does not generally help, can overflow some buffers also with `\0`
- Can remove some critical functions from (reduced) libc
- Problems:
  - Can break functionality
  - What functions exactly can cause problems. . . ?

Return Oriented Programming (ROP)

- We do not have to return to libc functions
- Can also return to arbitrary addresses (e.g., the `pop-retq` gadget)
- Can chain such returns, if each targeted block ends in `return`

Return Oriented Programming (ROP)

- We do not have to return to libc functions
- Can also return to arbitrary addresses (e.g., the `pop-retq` gadget)
- Can chain such returns, if each targeted block ends in `return`
- Attack idea: Collect pieces of code from binary (each ending in `return`)
- Chain these pieces to an attack program

Return Oriented Programming (ROP)

- We do not have to return to libc functions
- Can also return to arbitrary addresses (e.g., the `pop-retq` gadget)
- Can chain such returns, if each targeted block ends in `return`
- Attack idea: Collect pieces of code from binary (each ending in `return`)
- Chain these pieces to an attack program
- This idea is called *return-oriented programming*
- Concept introduced by Shacham in 2007
- ACM CCS 2017 Test of Time Award

Return Oriented Programming (ROP)

- We do not have to return to libc functions
- Can also return to arbitrary addresses (e.g., the `pop-retq` gadget)
- Can chain such returns, if each targeted block ends in `return`
- Attack idea: Collect pieces of code from binary (each ending in `return`)
- Chain these pieces to an attack program
- This idea is called *return-oriented programming*
- Concept introduced by Shacham in 2007
- ACM CCS 2017 Test of Time Award
- Collected pieces of code are called *gadgets*
- Attacker now has to program with "gadget-instructions"

Return Oriented Programming (ROP)

- We do not have to return to libc functions
- Can also return to arbitrary addresses (e.g., the `pop-retq` gadget)
- Can chain such returns, if each targeted block ends in `return`
- Attack idea: Collect pieces of code from binary (each ending in `return`)
- Chain these pieces to an attack program
- This idea is called *return-oriented programming*
- Concept introduced by Shacham in 2007
- ACM CCS 2017 Test of Time Award
- Collected pieces of code are called *gadgets*
- Attacker now has to program with "gadget-instructions"
- Slight generalization: Can also use gadgets ending in jumps

Return Oriented Programming (ROP)

- We do not have to return to libc functions
- Can also return to arbitrary addresses (e.g., the `pop-retq` gadget)
- Can chain such returns, if each targeted block ends in `return`
- Attack idea: Collect pieces of code from binary (each ending in `return`)
- Chain these pieces to an attack program
- This idea is called *return-oriented programming*
- Concept introduced by Shacham in 2007
- ACM CCS 2017 Test of Time Award
- Collected pieces of code are called *gadgets*
- Attacker now has to program with "gadget-instructions"
- Slight generalization: Can also use gadgets ending in jumps
- Important concept: can obtain *malicious computation* without *malicious code*!

Return Oriented Programming (ROP)

- We do not have to return to libc functions
- Can also return to arbitrary addresses (e.g., the `pop-retq` gadget)
- Can chain such returns, if each targeted block ends in `return`
- Attack idea: Collect pieces of code from binary (each ending in `return`)
- Chain these pieces to an attack program
- This idea is called *return-oriented programming*
- Concept introduced by Shacham in 2007
- ACM CCS 2017 Test of Time Award
- Collected pieces of code are called *gadgets*
- Attacker now has to program with "gadget-instructions"
- Slight generalization: Can also use gadgets ending in jumps
- Important concept: can obtain *malicious computation* without *malicious code*!
- Searching for gadgets (and to some extent chaining) can be automated

ROP: Example

vulnfunc()

```
...
retq
```

0xfeedface

```
...
xor %rax, %rax
retq
```

0xcafebabe

```
...
pop %rdi
retq
```

0xdeadbeef

```
...
mov %rdx, %rax
pop %rsi
retq
```

(corrupted) stack

0x7f1229d0f4a0 (execlp)
0x7f1229dd9f20 ("/bin/sh")
0xdeadbeef
0xfeedface
0x7f1229dd9f20 ("/bin/sh")
0xcafebabe
0x4141414141414141

registers

rax	unknown
rdx	unknown
rdi	unknown
rsi	unknown

ROP: Example

vulnfunc()

```
...
retq
```

0xfeedface

```
...
xor %rax, %rax
retq
```

0xcafebabe

```
...
pop %rdi
retq
```

0xdeadbeef

```
...
mov %rdx, %rax
pop %rsi
retq
```

(corrupted) stack

0x7f1229d0f4a0 (execlp)
0x7f1229dd9f20 ("/bin/sh")
0xdeadbeef
0xfeedface
0x7f1229dd9f20 ("/bin/sh")
0xcafebabe
0x4141414141414141

registers

rax	unknown
rdx	unknown
rdi	unknown
rsi	unknown

ROP: Example

vulnfunc()

```
...
retq
```

0xfeedface

```
...
xor %rax, %rax
retq
```

0xcafebabe

```
...
pop %rdi
retq
```

0xdeadbeef

```
...
mov %rdx, %rax
pop %rsi
retq
```

(corrupted) stack

0x7f1229d0f4a0 (execlp)
0x7f1229dd9f20 ("/bin/sh")
0xdeadbeef
0xfeedface
0x7f1229dd9f20 ("/bin/sh")
0xcafebabe
0x4141414141414141

registers

rax	unknown
rdx	unknown
rdi	unknown
rsi	unknown

ROP: Example

vulnfunc()

```
...
retq
```

0xfeedface

```
...
xor %rax, %rax
retq
```

0xcafebabe

```
...
pop %rdi
retq
```

0xdeadbeef

```
...
mov %rdx, %rax
pop %rsi
retq
```

(corrupted) stack

0x7f1229d0f4a0 (execlp)
0x7f1229dd9f20 ("/bin/sh")
0xdeadbeef
0xfeedface
0x7f1229dd9f20 ("/bin/sh")
0xcafebabe
0x4141414141414141

registers

rax	unknown
rdx	unknown
rdi	0x7f1229dd9f20
rsi	unknown

ROP: Example

vulnfunc()

```
...
retq
```

0xfeedface

```
...
xor %rax, %rax
retq
```

0xcafebabe

```
...
pop %rdi
retq
```

0xdeadbeef

```
...
mov %rdx, %rax
pop %rsi
retq
```

(corrupted) stack

0x7f1229d0f4a0 (execlp)
0x7f1229dd9f20 ("/bin/sh")
0xdeadbeef
0xfeedface
0x7f1229dd9f20 ("/bin/sh")
0xcafebabe
0x4141414141414141

registers

rax	unknown
rdx	unknown
rdi	0x7f1229dd9f20
rsi	unknown

ROP: Example

vulnfunc()

```
...
retq
```

0xfeedface

```
...
xor %rax, %rax
retq
```

0xcafebabe

```
...
pop %rdi
retq
```

0xdeadbeef

```
...
mov %rdx, %rax
pop %rsi
retq
```

(corrupted) stack

0x7f1229d0f4a0 (execlp)
0x7f1229dd9f20 ("/bin/sh")
0xdeadbeef
0xfeedface
0x7f1229dd9f20 ("/bin/sh")
0xcafebabe
0x4141414141414141

registers

rax	0x0
rdx	unknown
rdi	0x7f1229dd9f20
rsi	unknown

ROP: Example

vulnfunc()

```
...
retq
```

0xfeedface

```
...
xor %rax, %rax
retq
```

0xcafebabe

```
...
pop %rdi
retq
```

0xdeadbeef

```
...
mov %rdx, %rax
pop %rsi
retq
```

(corrupted) stack

0x7f1229d0f4a0 (execlp)
0x7f1229dd9f20 ("/bin/sh")
0xdeadbeef
0xfeedface
0x7f1229dd9f20 ("/bin/sh")
0xcafebabe
0x4141414141414141

registers

rax	0x0
rdx	unknown
rdi	0x7f1229dd9f20
rsi	unknown

ROP: Example

vulnfunc()

```
...
retq
```

0xfeedface

```
...
xor %rax, %rax
retq
```

0xcafebabe

```
...
pop %rdi
retq
```

0xdeadbeef

```
...
mov %rdx, %rax
pop %rsi
retq
```

(corrupted) stack

0x7f1229d0f4a0 (execlp)
0x7f1229dd9f20 ("/bin/sh")
0xdeadbeef
0xfeedface
0x7f1229dd9f20 ("/bin/sh")
0xcafebabe
0x4141414141414141

registers

rax	0x0
rdx	0x0
rdi	0x7f1229dd9f20
rsi	unknown

ROP: Example

vulnfunc()

```
...
retq
```

0xfeedface

```
...
xor %rax, %rax
retq
```

0xcafebabe

```
...
pop %rdi
retq
```

0xdeadbeef

```
...
mov %rdx, %rax
pop %rsi
retq
```

(corrupted) stack

0x7f1229d0f4a0 (execlp)
0x7f1229dd9f20 ("/bin/sh")
0xdeadbeef
0xfeedface
0x7f1229dd9f20 ("/bin/sh")
0xcafebabe
0x4141414141414141

registers

rax	0x0
rdx	0x0
rdi	0x7f1229dd9f20
rsi	0x7f1229dd9f20

ROP: Example

vulnfunc()

```
...
retq
```

0xfeedface

```
...
xor %rax, %rax
retq
```

0xcafebabe

```
...
pop %rdi
retq
```

0xdeadbeef

```
...
mov %rdx, %rax
pop %rsi
retq
```

(corrupted) stack

`0x7f1229d0f4a0 (execlp)`
`0x7f1229dd9f20 ("/bin/sh")`
`0xdeadbeef`
`0xfeedface`
`0x7f1229dd9f20 ("/bin/sh")`
`0xcafebabe`
`0x4141414141414141`

registers

`rax`	`0x0`
`rdx`	`0x0`
`rdi`	`0x7f1229dd9f20`
`rsi`	`0x7f1229dd9f20`

Will now jump to `execlp` with arguments in `rdi`, `rsi`, `rdx`
i.e. `execlp("/bin/sh", "/bin/sh", NULL);`

ASLR

- Return to libc and ROP *need to know the addresses of code*
- Idea: randomize position of dynamic libraries

ASLR

- Return to libc and ROP *need to know the addresses of code*
- Idea: randomize position of dynamic libraries
- This approach is called *address space layout randomization (ASLR)*
- Does not only randomize position of dynamic libraries, but also:
  - position of stack
  - position of data segment
  - position of heap

ASLR

- Return to libc and ROP *need to know the addresses of code*
- Idea: randomize position of dynamic libraries
- This approach is called *address space layout randomization (ASLR)*
- Does not only randomize position of dynamic libraries, but also:
  - position of stack
  - position of data segment
  - position of heap
- To also randomize the position of the binary itself need to use `gcc -fpie`
- `pie` stands for "position independent executable"

ASLR

- Return to libc and ROP *need to know the addresses of code*
- Idea: randomize position of dynamic libraries
- This approach is called *address space layout randomization (ASLR)*
- Does not only randomize position of dynamic libraries, but also:
  - position of stack
  - position of data segment
  - position of heap
- To also randomize the position of the binary itself need to use `gcc -fpie`
- `pie` stands for "position independent executable"
- Disable ASLR in Linux:
  `echo 0 > /proc/sys/kernel/randomize_va_space`
  or boot with parameter `norandmaps`

ASLR

- Return to libc and ROP *need to know the addresses of code*
- Idea: randomize position of dynamic libraries
- This approach is called *address space layout randomization (ASLR)*
- Does not only randomize position of dynamic libraries, but also:
  - position of stack
  - position of data segment
  - position of heap
- To also randomize the position of the binary itself need to use
  `gcc -fpie`
- `pie` stands for "position independent executable"
- Disable ASLR in Linux:
  `echo 0 > /proc/sys/kernel/randomize_va_space`
  or boot with parameter `norandmaps`
- Disable ASLR for one process:
  ``setarch `uname -m` -R PROGRAMNAME``

Attacks against ASLR

- ASLR is generally effective as a defense

Attacks against ASLR

- ASLR is generally effective as a defense
- Problem if address of one instruction leaks to the attacker:
  - Format-string attacks

Attacks against ASLR

- ASLR is generally effective as a defense
- Problem if address of one instruction leaks to the attacker:
  - Format-string attacks
  - Using overflows to overwrite null-termination

Attacks against ASLR

- ASLR is generally effective as a defense
- Problem if address of one instruction leaks to the attacker:
  - Format-string attacks
  - Using overflows to overwrite null-termination
  - Memory content written to disk

Attacks against ASLR

- ASLR is generally effective as a defense
- Problem if address of one instruction leaks to the attacker:
  - Format-string attacks
  - Using overflows to overwrite null-termination
  - Memory content written to disk
  - **All** libraries **must** be randomized
    - Can ROP on a non-randomized library

Attacks against ASLR

- ASLR is generally effective as a defense
- Problem if address of one instruction leaks to the attacker:
  - Format-string attacks
  - Using overflows to overwrite null-termination
  - Memory content written to disk
  - **All** libraries **must** be randomized
    - Can ROP on a non-randomized library
    - For a while, `linux-gate.so.1` was not randomized

Attacks against ASLR

- ASLR is generally effective as a defense
- Problem if address of one instruction leaks to the attacker:
  - Format-string attacks
  - Using overflows to overwrite null-termination
  - Memory content written to disk
  - **All** libraries **must** be randomized
    - Can ROP on a non-randomized library
    - For a while, `linux-gate.so.1` was not randomized
  - ...

Attacks against ASLR

- ASLR is generally effective as a defense
- Problem if address of one instruction leaks to the attacker:
  - Format-string attacks
  - Using overflows to overwrite null-termination
  - Memory content written to disk
  - **All** libraries **must** be randomized
    - Can ROP on a non-randomized library
    - For a while, `linux-gate.so.1` was not randomized
  - ...
- Problem on 32-bit machines: not enough entropy

Attacks against ASLR

- ASLR is generally effective as a defense
- Problem if address of one instruction leaks to the attacker:
  - Format-string attacks
  - Using overflows to overwrite null-termination
  - Memory content written to disk
  - **All** libraries **must** be randomized
    - Can ROP on a non-randomized library
    - For a while, `linux-gate.so.1` was not randomized
  - ...
- Problem on 32-bit machines: not enough entropy
  - Cannot randomize lower 12 bits of address (that would break page alignment)

Attacks against ASLR

- ASLR is generally effective as a defense
- Problem if address of one instruction leaks to the attacker:
  - Format-string attacks
  - Using overflows to overwrite null-termination
  - Memory content written to disk
  - **All** libraries **must** be randomized
    - Can ROP on a non-randomized library
    - For a while, `linux-gate.so.1` was not randomized
  - ...
- Problem on 32-bit machines: not enough entropy
  - Cannot randomize lower 12 bits of address (that would break page alignment)
  - Cannot randomize upper 4 bits (limits capabilities of large memory mappings)

Attacks against ASLR

- ASLR is generally effective as a defense
- Problem if address of one instruction leaks to the attacker:
  - Format-string attacks
  - Using overflows to overwrite null-termination
  - Memory content written to disk
  - **All** libraries **must** be randomized
    - Can ROP on a non-randomized library
    - For a while, `linux-gate.so.1` was not randomized
  - ...
- Problem on 32-bit machines: not enough entropy
  - Cannot randomize lower 12 bits of address (that would break page alignment)
  - Cannot randomize upper 4 bits (limits capabilities of large memory mappings)
  - Result: only 16 bits of entropy (65536 possibilities)

Attacks against ASLR

- ASLR is generally effective as a defense
- Problem if address of one instruction leaks to the attacker:
  - Format-string attacks
  - Using overflows to overwrite null-termination
  - Memory content written to disk
  - **All** libraries **must** be randomized
    - Can ROP on a non-randomized library
    - For a while, `linux-gate.so.1` was not randomized
  - ...
- Problem on 32-bit machines: not enough entropy
  - Cannot randomize lower 12 bits of address (that would break page alignment)
  - Cannot randomize upper 4 bits (limits capabilities of large memory mappings)
  - Result: only 16 bits of entropy (65536 possibilities)
  - Shacham, Page, Pfaff, Goh, Modadugu, Boneh, 2004: brute-force attack that took 216 seconds on average

Spot the defect – Heartbleed

```
/* Process incoming message with the format
| hbtype | len | payload[0] .... payload [len-1] |
one byte two bytes len bytes payload */
unsigned char *p; // pointer to the incoming message
unsigned int len; // called payload in the original code
unsigned short hbtype;
hbtype = *p++;
 // Puts *p into hbtype
n2s(p, len);
 // Takes two bytes from p, and puts them in len
// This is the length of the payload
unsigned char* buffer = malloc(1 + 2 + len);
/* Enter response type, length and copy payload */
buffer++ = TLS1_HB_RESPONSE;
s2n(len, buffer);
 // takes 16-bit value len and puts it into two bytes
memcpy(buffer, p, len); // copy len bytes from p into buffer
```

Spot the defect – Heartbleed

```
/* Process incoming message with the format
| hbtype | len | payload[0] .... payload [len-1] |
one byte two bytes len bytes payload */
unsigned char *p; // pointer to the incoming message
unsigned int len; // called payload in the original code
unsigned short hbtype;
hbtype = *p++;
 // Puts *p into hbtype
n2s(p, len);
 // Takes two bytes from p, and puts them in len
// This is the length of the payload
unsigned char* buffer = malloc(1 + 2 + len);
/* Enter response type, length and copy payload */
buffer++ = TLS1_HB_RESPONSE;
s2n(len, buffer);
 // takes 16-bit value len and puts it into two bytes
memcpy(buffer, p, len); // copy len bytes from p into buffer
```

↖ Possible uninitialized data read

Spot the defect – Cloudbleed

```
// char* p is a pointer to a buffer containing the
//         incoming messages to be processed
// char* end is a pointer to the end of this buffer
....
// code inspecting *p, which increases p
....
if ( ++p == end ) goto _test_eof;
```

More secure code

```
....
if ( ++p >= end ) goto _test_eof;
```

How common are these problems?

Look at websites such as

- https://www.us-cert.gov/ncas/bulletins
- http://cve.mitre.org/
- http://www.securityfocus.com/vulnerabilities

How common are these problems?

Look at websites such as

- https://www.us-cert.gov/ncas/bulletins
- http://cve.mitre.org/
- http://www.securityfocus.com/vulnerabilities

Vulnerability descriptions that mention

- 'buffer'
- 'boundary condition error'
- 'lets remote users execute arbitrary code'
- or simply 'remote security vulnerability'

are often caused by buffer overflows. Some sites use the CWE (Common Weakness Enumeration) to classify vulnerabilities.

CWE classification

The CWE (Common Weakness Enumeration) provides a standardised classfication of security vulnerabilities https://cwe.mitre.org/ NB the classification is long (over 800 classes!) and confusing! Eg

- CWE-118 ... CWE-129, CWE-680, and CWE 787 are buffer errors
- CWE-822 ... CWE-835 and CWE-465 are pointer errors
- CWE-872 are integer-related issues

CWE classification

The CWE (Common Weakness Enumeration) provides a standardised classfication of security vulnerabilities https://cwe.mitre.org/ NB the classification is long (over 800 classes!) and confusing! Eg

- CWE-118 ... CWE-129, CWE-680, and CWE 787 are buffer errors
- CWE-822 ... CWE-835 and CWE-465 are pointer errors
- CWE-872 are integer-related issues

Have a look at

- `https://cwe.mitre.org/data/definitions/787.html` - buffer issues
- `https://cwe.mitre.org/data/definitions/465.html` - pointer issues
- `https://cwe.mitre.org/data/definitions/872.html` - integer issuess

Example vulnerable code

```
void m() {
  int x = 4;
  f(); // return_to_m
  printf ("x is %d", x);
}

void f() {
  int y = 7;
  g(); // return_to_f
  printf ("y+10 is %d", y+10);
}

void g() {
  char buf[80];
  gets(buf);    ← potential overflow of buf
  printf(buf);  ← potential format string attack
  gets(buf);    ← potential overflow of buf
}
```

Example vulnerable code

```
void m() {
  int x = 4;
  f(); // return_to_m
  printf ("x is %d", x);
}

void f() {
  int y = 7;
  g(); // return_to_f
  printf ("y+10 is %d", y+10);
}

void g() {
  char buf[80];
  gets(buf);      ← potential overflow of buf
  printf(buf);    ← potential format string attack
  gets(buf);      ← potential overflow of buf
}
```

An attacker could

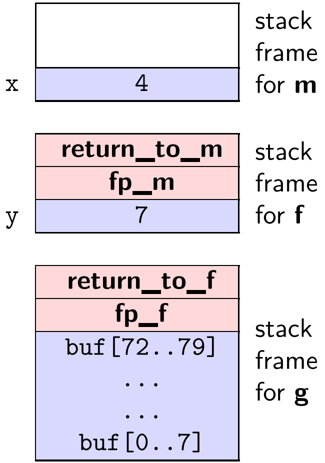
1. first inspect the stack using a malicious format string (entered in first gets and printed with `printf`)
2. then overflow buf to corrupt the stack (with the second `gets`)

Example vulnerable code

```
void m() {
  int x = 4;
  f(); // return_to_m
  printf ("x is %d", x);
}

void f() {
  int y = 7;
  g(); // return_to_f
  printf ("y+10 is %d", y+10);
}

void g() {
  char buf[80];
  gets(buf);
  printf(buf);
  gets(buf);
}
```



Normal execution

- After completing **g**

  execution continues with **f** from program point **return_to_f**

  This will print 17.

Normal execution

- After completing **g**

  execution continues with **f** from program point **return_to_f**

  This will print 17.
- After completing **f**

  execution continues with **main** from program point **return_to_m**

  This will print 4.

Normal execution

- After completing **g**

  execution continues with **f** from program point **return_to_f**

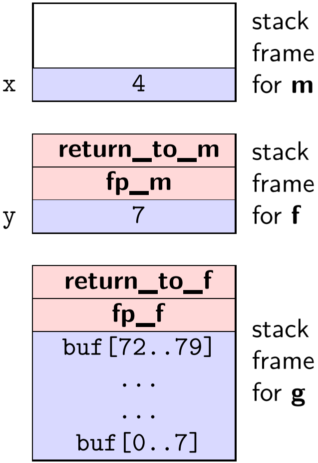
  This will print 17.
- After completing **f**

  execution continues with **main** from program point **return_to_m**

  This will print 4.

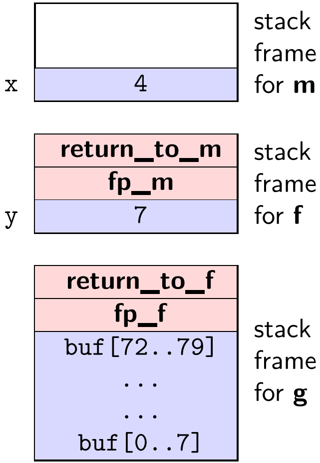
If we start smashing the stack different things can happen

Attack scenario 1



in **g()** we overflow **buf** to overwrite values of **x** or **y**.

Attack scenario 1



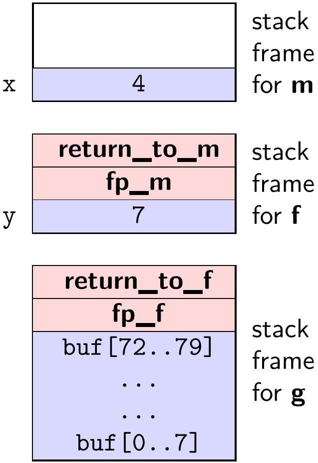
in **g()** we overflow **buf** to overwrite values of **x** or **y**.

- After completing **g**

  execution continues with **f** from program point **return_to_f**

  This will print whatever value we gave to **y** $+10$.

Attack scenario 1



in **g()** we overflow **buf** to overwrite values of **x** or **y**.

- After completing **g**

  execution continues with **f** from program point **return_to_f**

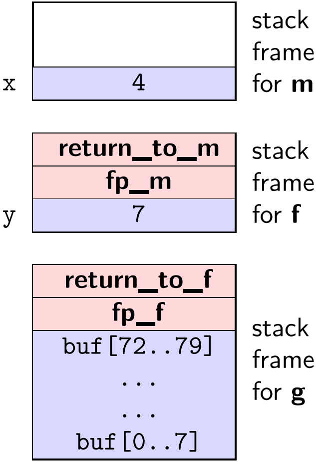
  This will print whatever value we gave to **y** +10.

- After completing **f**

  execution continues with **main** from program point **return_to_m**

  This will print whatever value we gave to **x**.

Attack scenario 1



in **g()** we overflow **buf** to overwrite values of **x** or **y**.

- After completing **g**

  execution continues with **f** from program point **return_to_f**

  This will print whatever value we gave to **y** +10.

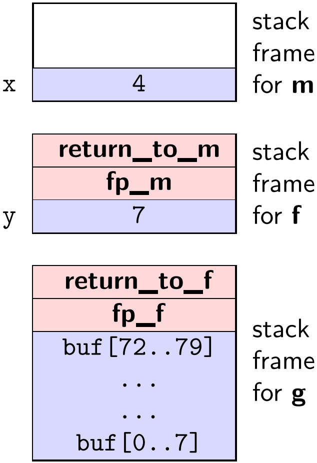
- After completing **f**

  execution continues with **main** from program point **return_to_m**

  This will print whatever value we gave to **x**.

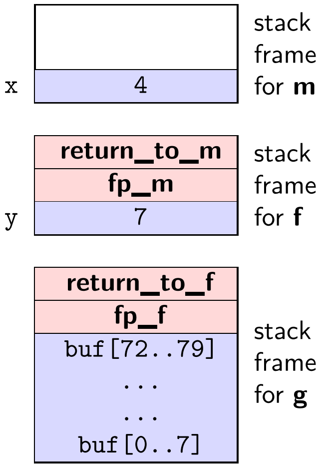
Of course, it is easier to overwrite local variables in the current frame than variables in 'lower' frames

Attack scenario 2



in **g()** we overflow **buf** to overwrite return address **return_to_f** with **return_to_m**

Attack scenario 2



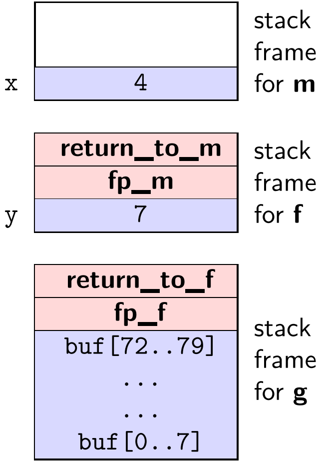
in **g()** we overflow **buf** to overwrite return address **return_to_f** with **return_to_m**

- After completing **g**

  execution continues with **m** instead of **f** but with **f**'s stack frame.

  This will print 7.

Attack scenario 2



in **g()** we overflow **buf** to overwrite return address **return_to_f** with **return_to_m**

- After completing **g**

  execution continues with **m** instead of **f** but with **f**'s stack frame.

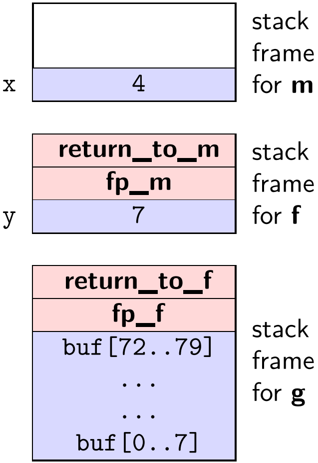
  This will print 7.

- After completing **m**

  execution continues with **m**

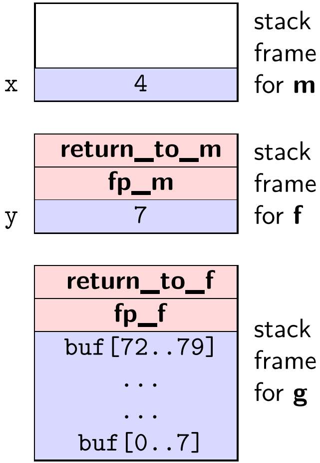
  This will print 4;

Attack scenario 3



in **g()** we overflow **buf** to overwrite frame pointer **fp_f** with **fp_m**.

Attack scenario 3



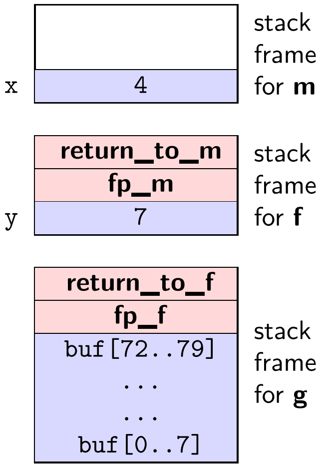
in **g()** we overflow **buf** to overwrite frame pointer **fp_f** with **fp_m**.

- After completing **g**

  execution continues with **f** but with **m**'s stack frame.

  This will print 14.

Attack scenario 3



in **g()** we overflow **buf** to overwrite frame pointer **fp_f** with **fp_m**.

- After completing **g**

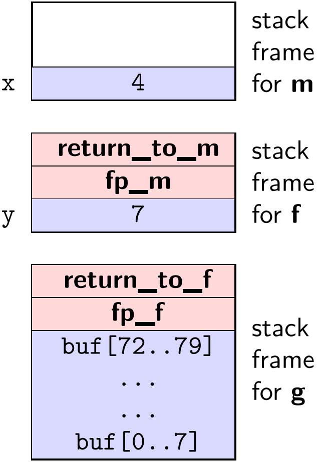
  execution continues with **f** but with **m**'s stack frame.

  This will print 14.

- After completing **f**

  execution continues with whatever code called **m**.

Attack scenario 3



in **g()** we overflow **buf** to overwrite frame pointer **fp_f** with **fp_m**.

- After completing **g**

  execution continues with **f** but with **m**'s stack frame.

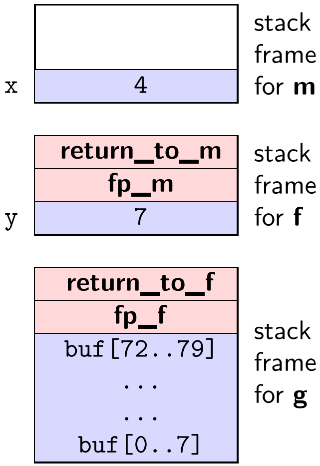
  This will print 14.

- After completing **f**

  execution continues with whatever code called **m**.

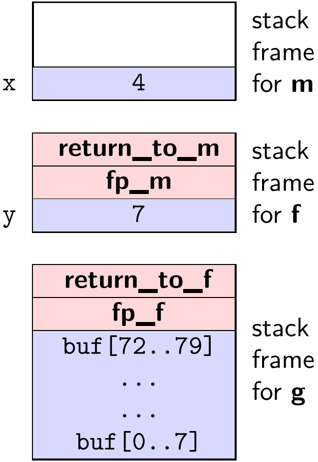
So we never finish the function call **m**, the remaining part of the code (after the call to **f**) will never be executed.

Attack scenario 4



in **g()** we overflow **buf** to overwrite frame pointer **fp_f** with **fp_g**.

Attack scenario 4



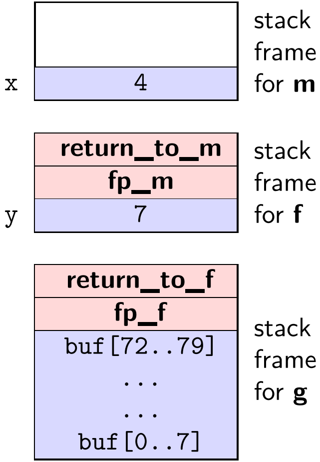
in **g()** we overflow **buf** to overwrite frame pointer **fp_f** with **fp_g**.

- After completing **g**

  execution continues with **f**
  but with **g**'s stack frame.

  This will print (some bytes of **buf** +10).

Attack scenario 4



in **g()** we overflow **buf** to overwrite frame pointer **fp_f** with **fp_g**.

- After completing **g**

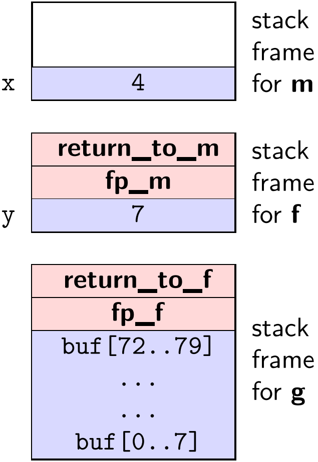
  execution continues with **f**
  but with **g**'s stack frame.

  This will print (some bytes of **buf** +10).

- After completing **f**

  execution might continue with **f**, again with **g**'s stack frame, repeating this forever.

Attack scenario 4



in **g()** we overflow **buf** to overwrite frame pointer **fp_f** with **fp_g**.

- After completing **g**

  execution continues with **f** but with **g**'s stack frame.

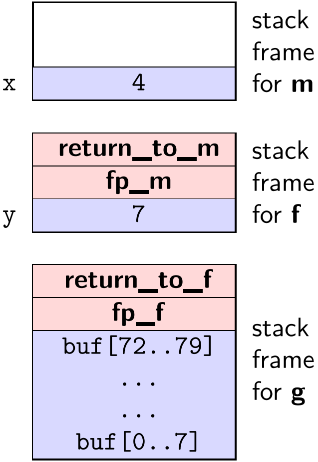
  This will print (some bytes of **buf** +10).

- After completing **f**

  execution might continue with **f**, again with **g**'s stack frame, repeating this forever.

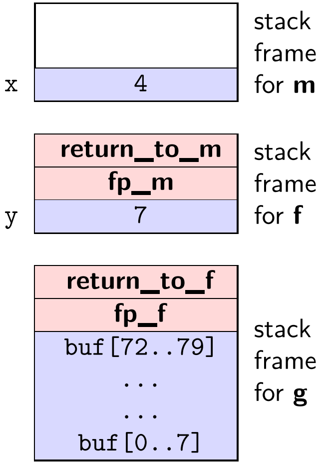
This depends on whether the compiled code looks up values from the top of **g**'s stack frame, or the bottom of **g**'s stack frame. In the latter case the code will jump to some code depending on the contents of **buf**.

Attack scenario 5



in **g()** we overflow **buf** to overwrite frame pointer **fp_f** with some pointer into **buf**.

Attack scenario 5



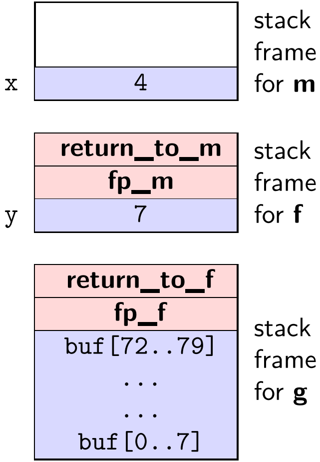
in **g()** we overflow **buf** to overwrite frame pointer **fp_f** with some pointer into **buf**.

- After completing **g**

  execution continues with **f**
  but with part of **buf** as stack frame.

  This will print (some bytes of **buf** +10).

Attack scenario 5



in **g()** we overflow **buf** to overwrite frame pointer **fp_f** with some pointer into **buf**.

- After completing **g**

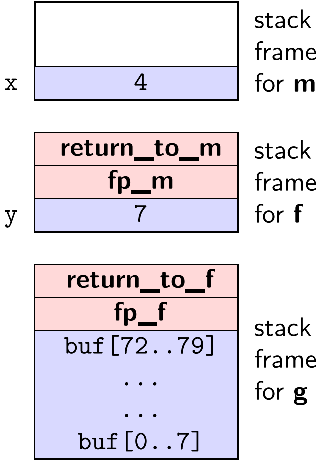
  execution continues with **f**
  but with part of **buf** as stack frame.

  This will print (some bytes of **buf** +10).

- After completing **f**

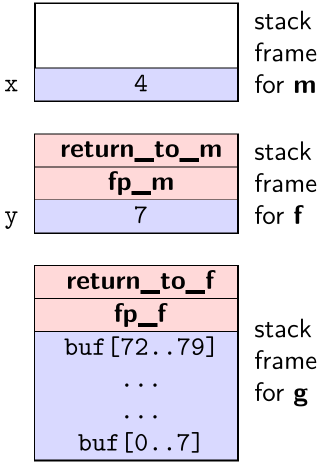
  execution continues with an address and frame pointer taken from **buf**

Attack scenario 6



in **g()** we overflow **buf** to overwrite the return address **return_to_f** to point in some code somewhere, and the framepointer to point inside **buf**.

Attack scenario 6



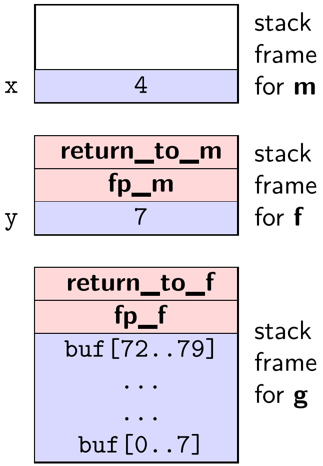
in **g()** we overflow **buf** to overwrite the return address **return_to_f** to point in some code somewhere, and the framepointer to point inside **buf**.

- After completing **g**

  execution continues executing that code using part of **buf** as stack frame.

  This can do all sorts of things!
  If we have enough code to choose from, this can do anything we want.

Attack scenario 6



in **g()** we overflow **buf** to overwrite the return address **return_to_f** to point in some code somewhere, and the framepointer to point inside **buf**.

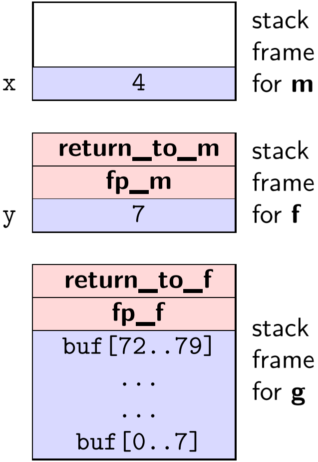
- After completing **g**

  execution continues executing that code using part of **buf** as stack frame.

  This can do all sorts of things!
  If we have enough code to choose from, this can do anything we want.

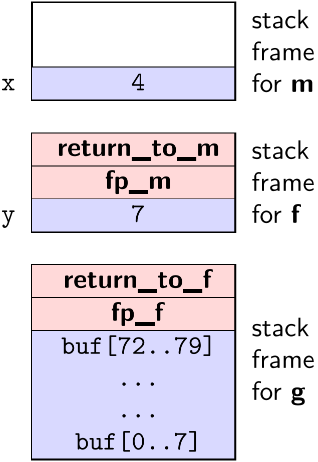
Often the address of a function in libc is used, in what is called a return-to-libc attack.

Attack scenario 7



in **g()** we overflow **buf** to overwrite the return address **return_to_f** to point inside **buf**

Attack scenario 7

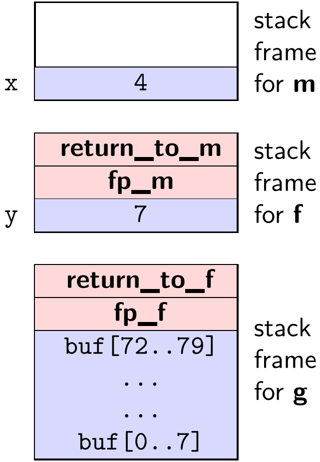


in **g()** we overflow **buf** to overwrite the return address **return_to_f** to point inside **buf**

- After completing **g**

  execution continues with whatever code (aka shell code) was written in **buf**, using **f**'s stack frame. This can do anything we want.

Attack scenario 7



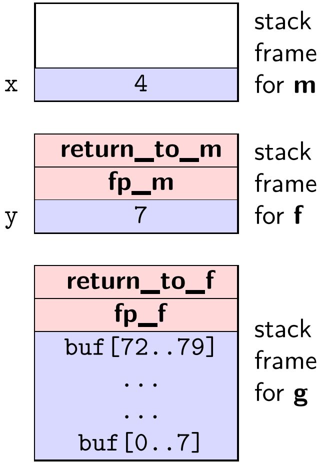
in **g()** we overflow **buf** to overwrite the return address **return_to_f** to point inside **buf**

- After completing **g**

  execution continues with whatever code (aka shell code) was written in **buf**, using **f**'s stack frame. This can do anything we want.

This is the classic buffer overflow attack discussed last week

Attack scenario 7



in **g()** we overflow **buf** to overwrite the return address **return_to_f** to point inside **buf**

- After completing **g**

  execution continues with whatever code (aka shell code) was written in **buf**, using **f**'s stack frame. This can do anything we want.

This is the classic buffer overflow attack discussed last week

- The attack requires that the computer (OS + hardware) can be tricked into executing data allocated on the stack. Many systems will no longer execute data (code) on the stack or on the heap (last week).