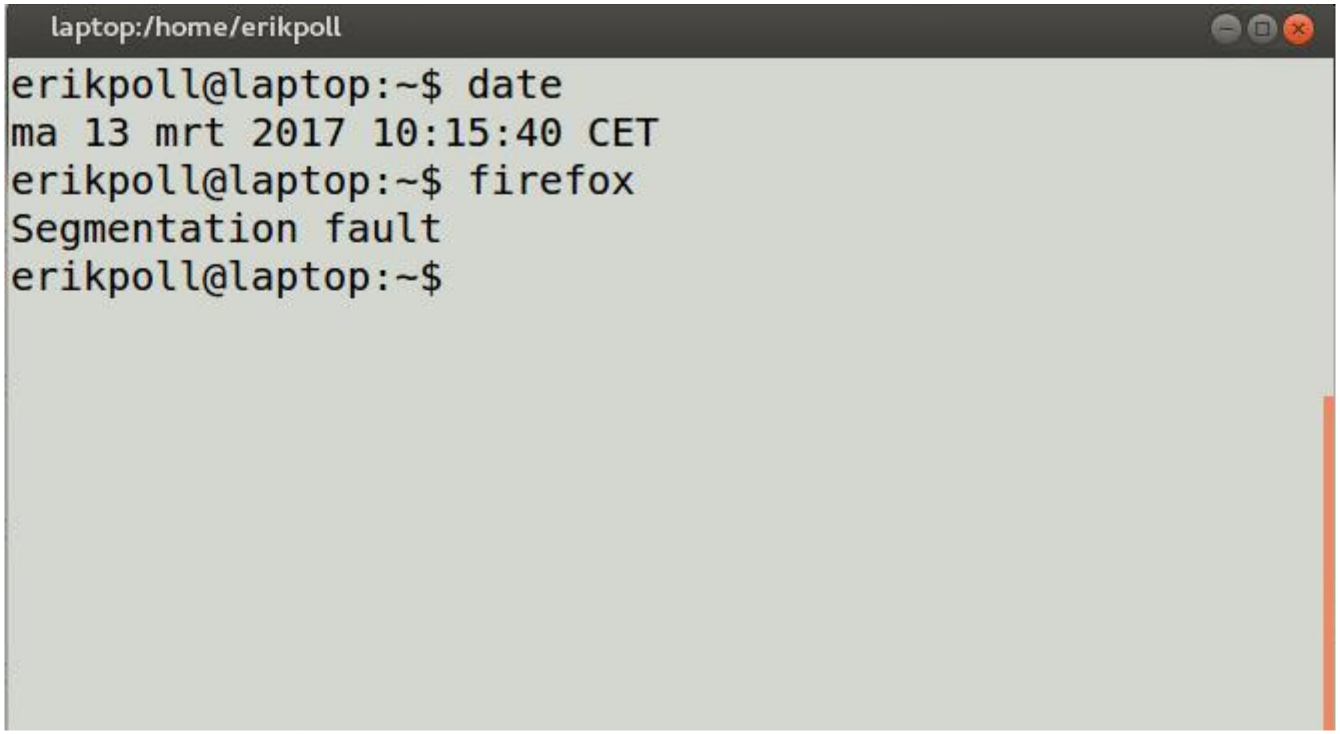
Attacking the stack (continued)

Firefox bugs this morning



```
erikpoll@laptop:~$ date
ma 13 mrt 2017 10:15:40 CET
erikpoll@laptop:~$ firefox
Segmentation fault
erikpoll@laptop:~$
```

Last week

- Using buffer overruns or format string attacks to read out or corrupt the stack, esp. the control data on the stack:

  frame pointers and return addresses
- A classic buffer overflow:
  1. first insert malicious shell code into some buffer on the stack
  2. then overwrite return address on the stack to jump to that code

  This is also called *smashing the stack*
- More generally, any read or write out-of-bounds, use of a stale pointer, or reading of uninitialized memory, ... can be a security problem

Today:

- some variations of messing with frame pointers & return addresses
- some defenses

Spot the defect

```
/*   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 OVERRUN
```

Spot the defect – Cloudbleed

Vulnerable code

```
// char* p is a pointer to a buffer containing the incoming messages to be process
// 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

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

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

warning: potential exam questions coming up

example vulnerable code

```
m(){
 int x = 4;
 f(); // return_to_m
 printf (“x is %i”, x);}

f(){
 int y = 7;
 g(); // return_to_f
 printf (“y+10 is %i”, y+10);}

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

example vulnerable code

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

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

g(){
 char buf[80];
 gets(buf);
 printf(buf);
 gets(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**)

potential overflow of **buf**

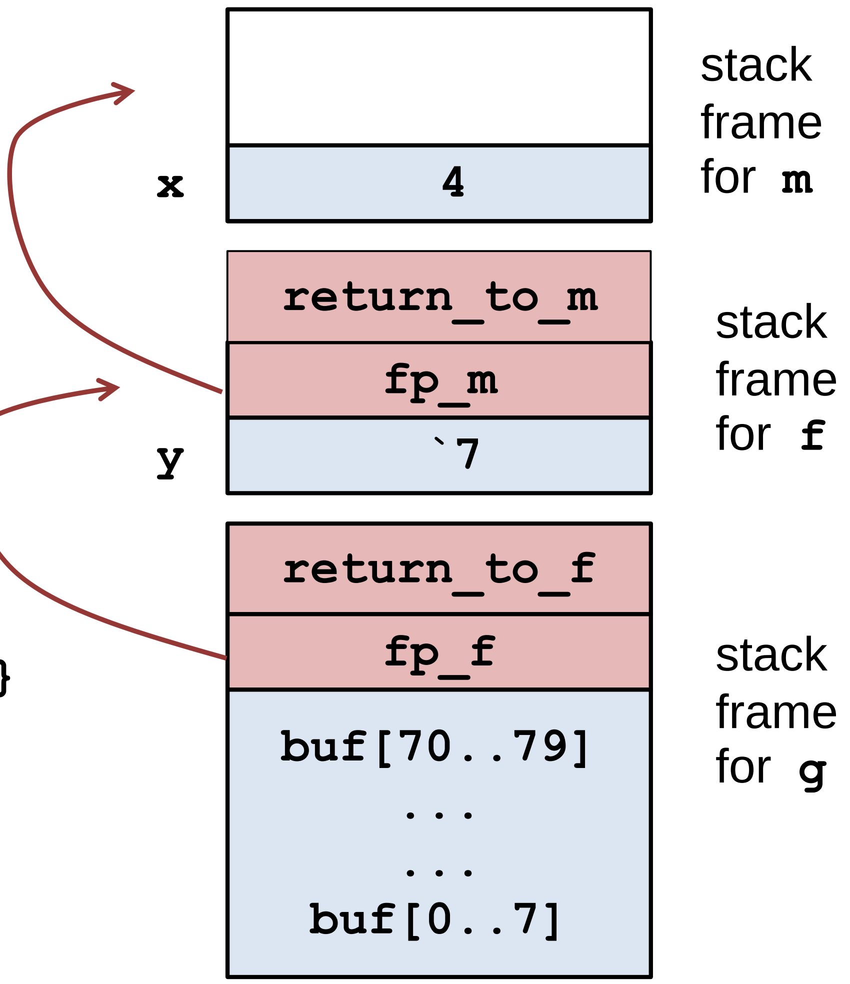
potential format string attack

example vulnerable code

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

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

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.

- 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`**.

- 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 **`m`** 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`**

- 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`**

- 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`**

- 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 for ever.

  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`**

- After completing **`g`**
  execution continues with **`f`**
  but with part of **`buf`** as stack frame.

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

- After completing **`f`**
  not clear what will happen...

Attack scenario 6

In **`g()`** we overflow **`buf`** to overwrite the return address **`return_to_f`** to point in some code somewhere, and the frame pointer 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 a return address in some library routine in libc is used,
in what is then called a return-to-libc attack.

Attack scenario 7

In **`g()`** we overflow **`buf`** to overwrite the return address 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

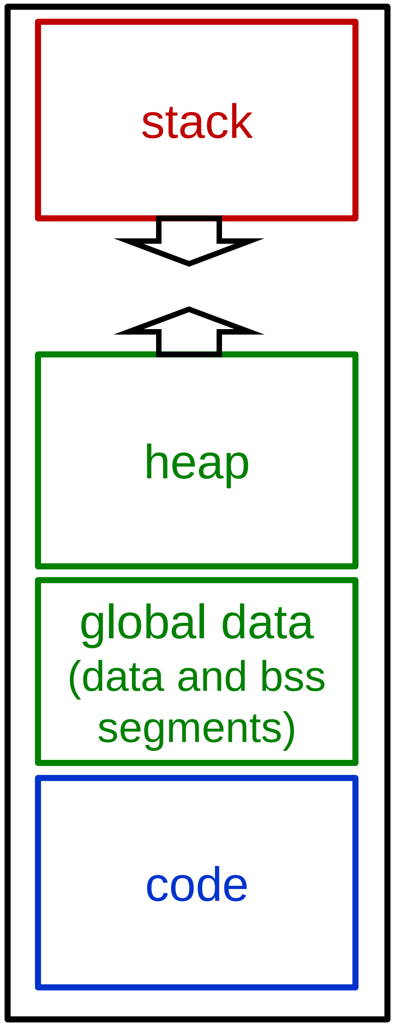
- You could also overwrite **`sp_f`** and supply the attack code with a fake stack frame, but typically the shell code won't need a stack frame
- This attack requires that the computer (OS+ hardware) can be tricked into executing data allocated on the stack. Many systems will no longer execute data (code) on the stack or on the heap – see slide 29 & further on

Memory segments revisited

Normally (always?) the program counter should point somewhere in the code segment

The attack scenarios discussed in these slides only involved overflowing buffers on the stack.

Buffers allocated on the heap or as global variables can also be overflowed to change program behaviour, but to mess with return addresses or frame pointers we need to overflow on the stack



Defense mechanisms

How do we stop this from happening?

Thanks to iseclab and syssec course repository for some of thse slides

Defenses

Different strategies:

- **Prevent**
- **Detect**
- **React**

Ideally, you'd like to *prevent* problems, but typically you cannot,.
The best we can do

- *make it harder* for the attacker
- *mitigate* the potential impact
- *detect* attacks and *react*

prevent

detect

react

Defenses at different levels

- At program level
  - to prevent attacks by removing the vulnerabilities
- At compiler level
  - to detect and block exploit attempts
- At operating system level
  - to make the exploitation much more difficult

First of all: the human factor

- The main cause of buffer overflows is not C, it is bad programmers ;-)
  - educate programmers how to write secure code
    - eg. everyone should know never to use **`gets`**
  - test programs with a focus on security issues

- Switch to more secure library functions
  - Standard Library: strncpy, strncat, ...
  - BDS's strlcpy, strlcat (boundary safe)
  - LibSafe: wrapper around a set of potentially “dangerous” libc functions
  - ContraPolice: libc extension to prevent heap overflow

Runtime checking: Libsafe

- Intercepts calls to dangerous functions that manipulate strings
  `strcpy, strcat, getwd, gets, [vf]scanf, realpath, [v]sprint,...`
- Uses frame pointers to estimate upper bound for buffer sizes:
  `buffer size <  |EBP - buff address|`
- Adds runtime checks to make sure that any buffer overflows are contained within the current stack frame

Program Level: dynamic analysis tools

Tools that instrument code with runtime checks that try to detect bugs, incl. buffer overflows, memory leaks, dangling pointers,

- ValGrind
- Purify
- AddressSanitizer
- Clang
- ....

Program Level: static analysis tools

Tools that analyse code at compile-time to spot potential buffer overflows.

- Ccured
- Flawfinder
- Insure++
- CodeWizard
- Cigital ITS4
- Cqual
- Microsoft PREfast/PREfix
- Pscan
- RATS
- Fortify

Can you attack this function with a buffer overflow?

```
void f(....){
   long canary = CANARY_VALUE; // initialise canary
   ...
   ... // overflows a buffer on the stack
   ...
   if (canary != CANARY_VALUE) {
      exit(CANARY_DEAD); // abort with error code
   }
}
```

Stack protection with canaries

Goal:

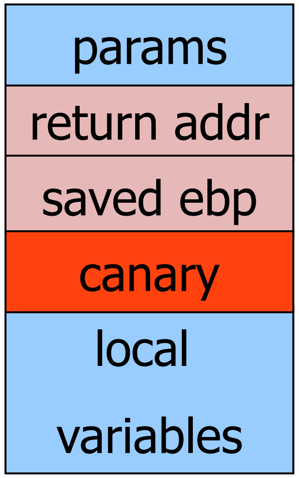
detect if some buffer overflow wrote over the boundary of the stack frame, corrupting the control data

How this works: the compiler inserts code to

– add a "canary" value between the local variables and the saved EBP

– at the end of the function, check that the canary is “still alive”

– a changed canary value means that a buffer preceding it in memory has been overflowed - ie. the stack has been smashed!



Stack canaries

You could add your own code to add & check stack canaries,
by adding the lines below to the beginning and end of functions

```
void f(....){
  long canary = CANARY_VALUE; // initialise canary
  ...
  ...
  if (canary != CANARY_VALUE) {
     exit(CANARY_DEAD); // abort with error code
  }
}
```

It is easier & better to let the compiler add this code for you.
That can cover all the points where the function returns.
gcc does this with the `-fstack-protector` option

Stack canaries

A clever attacker can try to put the correct canary value back.
Tricks to make this harder

- Generate a random canary value each time a program starts,
  so the attacker cannot predict the value it

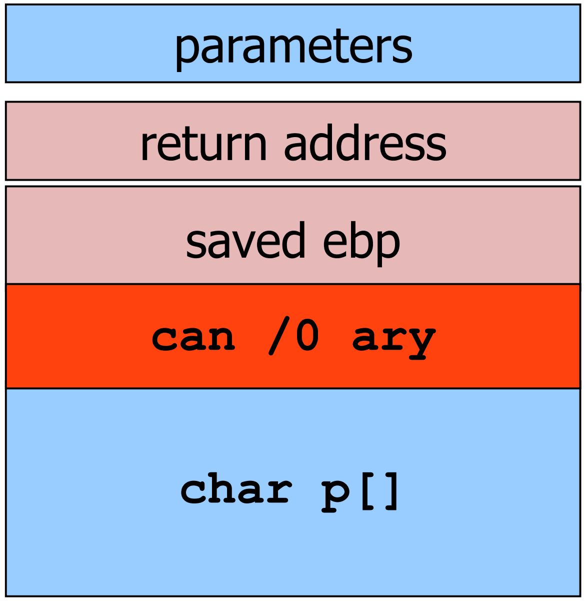
- Make sure there is a null-terminator, ie. the character `\0’,a somewhere in the middle of the canary value.

Including the null terminator \0 in the canary

Can you over overflowing the array p with some string copying function, corrupting the return address but leaving the canary intact?

No,you'd need two operations:

1. one to change the return address, and put last part of canary, `ary`, back. This will remove the `/0`.
2. a second one to write the first part of the canary, `can /0`, back



Question: how many overflows would be needed if there are 4 null terminators in the canary?

Stack canaries

A clever attacker could try to put the correct canary value back.

Tricks to make this harder

- Generate a random canary value each time a program starts,

  so the attacker cannot predict the value it
- Make sure there is a null-terminator, ie. the character `\0',a somewhere in the middle of the canary value.

  This makes it impossible for the attacker to write the canary value back using a standard string writing functions, as these functions will stop at null-terminators.
- Let the canary be the XOR of some "master" canary value and the return address on the stack.

  If the attacker then changes the return address, then the old canary value is no longer correct.

Windows 2003 Stack Protection

The subtle ways in which things can still go wrong...

Microsoft introduced stack canaries in 2003 in its compilers

- Enabled with /GS command line option
- When canary is corrupted, control is transferred to an exception handler
- Exception handler information is stored ... on the stack
  - http://www.securityfocus.com/bid/8522/info
- Countermeasure: register exception handlers, and don't trust exception handlers that are not registered or that are on the stack
- Attackers may still abuse existing handlers or point to exception handler outside the loaded module...

*N*on e*X*ecutable Stack (NX aka W⊕X)

- Does not block buffer overflows, but prevent the shellcode from being executed
  - can affect the execution of some programs that normally require to execute data on the stack
  - it makes use of hardware features such as the NX bit (IA-64, AMD64)

- Supported by many operating systems
  - MacOs X
  - Data Execution Prevention (DEP) on Windows
  - OpenBSD W^X
  - ExecShield and PAX patches in Linux

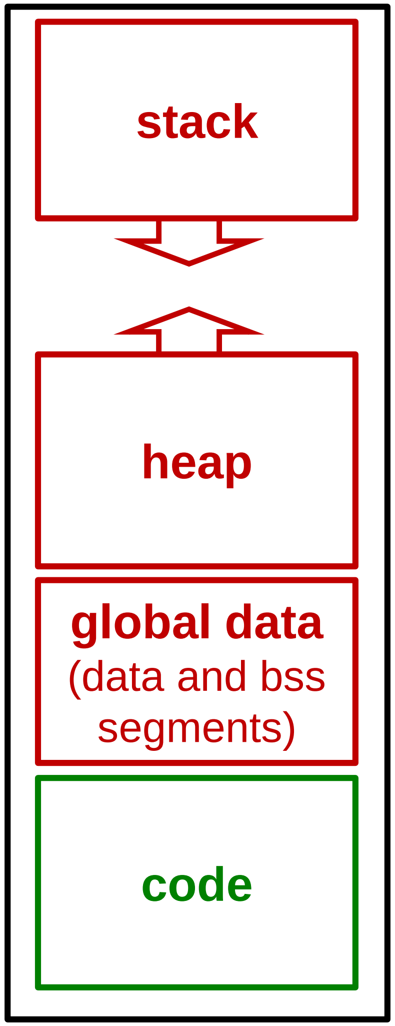
non-executable memory

Memory used by a process (program in execution) consists of different segments.

Program counter should point to the code segment,
not to heap, stack, or data segments.

Making these segments *non-executable* makes it impossible for attackers to execute their own malicious code that they manage to get into some stack- or heap-allocated buffer.

But *not* the attacks discussed earlier today, which involve jumping to existing code or corrupting frames



Doesn’t this solve the problem?

Stack canaries and non-executable stack make attacks harder,
but not impossible.

For example

- attacker may be able restore the canary values
- attacker may be able to jump to existing code, eg in return-to-libc attacks, defeating the non-executable stack protection
- other interesting targets for the attacker might not be protected by these measures, eg
  - data on the stack in the current frame
  - function pointers allocated on the stack or the heap

    (We will not go into function pointers, or expect you to know this for the exam)

The rest is not exam material

Defeating NX: return-to-libc attacks

Code *injection* attacks are no longer possible,
but code *reuse* attacks are...

So instead of jumping to own attack code in non-executable buffer
overflow the stack to jump to code that is already there,
esp. library code in **`libc`**

`libc` is a rich library that offers many possibilities for attacker,
eg. **`system(), exec(),`**
which provide the attacker with any functionality he wants...

reTURN oriented programMing (ROP)

Next stage in evolution of attacks, as people removed or protected dangerous library calls like **`system()`**

Instead of using entire library call, the attacker

- looks for gadgets, small snippets of code which end with a return, in the existing code bas,

```
        ... ; ... ; ... ; ret
```

- strings these gadgets together as subroutines to form a program that does what he wants

This turns out to be doable

- Most libraries contain enough gadgets to provide a Turing complete programming language
- An ROP compiler can then translate any code to a string of these gadgets

ASLR

- Introduce artificial diversity by randomly arranging the positions of key data areas:

  base of the executable, position of libraries, heap, and stack)

- typically, a varying offset added to the start of the stack, the heap, etc
  - ASLR (Address Space Layout Randomization)

This prevents the attacker from being able to easily predict target addresses