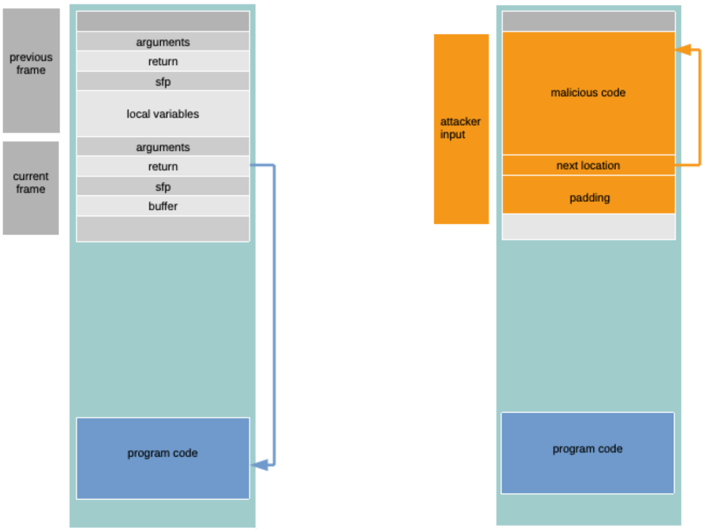
CS Revision Lecture 23, 24

- ## Lecture 23 - Buffer overrun attacks
  - ### Control hijacking
    - A buffer overflow can change the flow of execution of the program

      - load malicious code into memory
      - make %eip point to it
  - ### Shellcode injection
    - Goal:
      - "Spawn a shell" - will give the attacker general access to the system

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

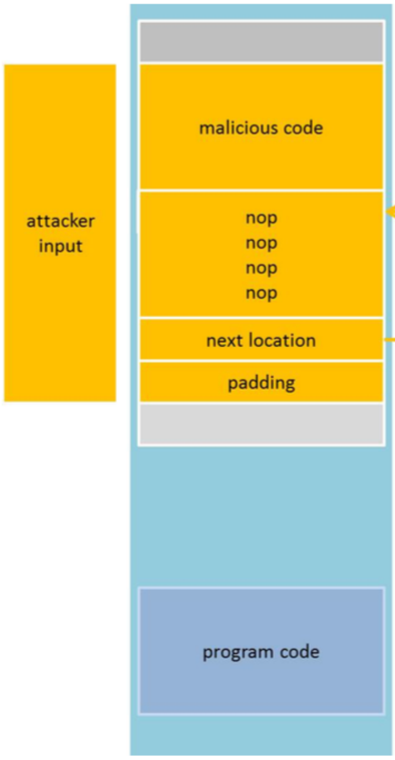
C code

```
"\x31\xc0"
"\x50"
"\x68" "//sh"
"\x68" "/bin"
"\x89\xe3"
"\x50"
...
```

Machine code
(part of attacker's input)

    - Must inject the machine code instructions (code ready to run)
    - the code cannot contain any zero bytes (printf, gets, strcpy will stop copying)
    - can't use the loader (we're injecting)

- The return address
  - Challenge: Find the address of the injected malicious code
    - If code accessible: we know how far is the overflowed variable from the saved %ebp
    - If code not accessible: try different possibilities! In a 32 bits memory space, there are 2^{32} possibilities
    - NOP sled
      - Guess approximate stack state when the function is called
      - insert many NOPs before Shell Code



- Buffer overrun opportunities
  - Unsafe libc functions
    - strcpy (char *dest, const char *src)
    - strcat (char *dest, const char *src)
    - gets (char *s)
    - scanf (const char *format, ...)
    - **Do not Check bounds of buffers they manipulate!!**

- Use safe functions

```
int read_stdi(void){
  char buf[128];
  int i;
  fgets(buf, sizeof(buf), stdin);
  i = atoi(buf);
  return i;
}
```

  - Your program is as secure as its programmer is cautious. It is now up to the programmer to include all the necessary checks in his program :- / and this is a tricky one
- Arithmetic overflows
  - Limitation related to the representation of integers in memory
  - In 32 bits architectures, signed integers are expresses in **two's compliment notation**
    - 0x00000000 - 0x7fffffff : positive numbers 0 -- $(2^{31} - 1)$
    - 0x80000000 - 0xffffffff : negative numbers $(-2^{31} + 1)$ -- (-1)
  - In 32 bits architectures, unsigned integers are only positive numbers 0x00000000 - 0xffffffff. Once the highest unsigned integer is reached, the next sequential integer wraps around zero.

```
# include <stdio.h>
int main(void){
  unsigned int num = 0xffffffff;
  printf(``num + 1 = 0x%x\n'', num + 1);
  return 0;
}
```

  - The output of this program is: **num + 1 = 0x0**
- Arithmetic overflow exploit (1)
  - Stack-based buffer overflow due to arithmetic overflow:

```
int catvars(char *buf1, char *buf2,
                  unsigned int len1, unsigned int len2){
  char mybuf[256];
  if((len1 + len2) > 256){
    return -1;
  }
  memcpy(mybuf, buf1, len1);
  memcpy(mybuf + len1, buf2, len2);
  do_some_stuff(mybuf);
  return 0;
}
```

- **Check can be bypassed by using suitable values for** len1 **and** len2, e.g. len1 = 0x00000103, len2 = 0xfffffffc, len1+len2 = 0x000000ff (decimal 255)

- Arithmetic overflow exploit (2)
  - Stack-based buffer overflow due to arithmetic overflow:

```
int catvars(char *buf, int len){
  char mybuf[256];
  if(len > 256){
    return -1;
  }
  memcpy(mybuf, buf, len);
  return 0;
}
```

  - memcpy(void *s1, const void *s2, size_t n); // size_t is unsigned
  - Check can be bypassed by using suitable values for len, e.g. len = -1 = 0xffffffff , will be interpreted as an unsigned integer encoding the value $2^{32} - 1$!
- Arithmetic overflow exploit (3)
  - Heap-based buffer overflow due to arithmetic overflow:
    - Memory **dynamically** allocated will persist across multiple function calls.
    - This memory is allocated on the heap segment.
    - Heap-based buffer overflows are more complex, and require understanding garbage collection and heap implementation.

```
int myfunction(int *array, int len){
  int *myarray, i;
  myarray = malloc(len * sizeof(int));
  if(myarray == NULL){
    return -1;
  }
  for(i = 0; i < len; i++){
    myarray[i] = array[i];
  }
  return myarray;
}
```

- **Can allocate a size 0 buffer for** myarray **by using suitable value for** len: len = 1073741824 , sizeof(int) = 4, len*sizeof(int) = 0

- The Ariane 5 Disaster
  - Attempt to store a value in an integer which is greater than the maximum value the integer can hold $\longrightarrow$ **the value will be truncated**

Ariane 5 rocket launch explosion due to integer overflow

- Format Strings
  - A format function takes a variable number of arguments, from which one is the so called format string
    - **Examples: fprintf, printf, ..., syslog, ..,**
      - **printf("The amount is %d pounds\n", amnt);**
  - The behaviour of the format function is controlled by the format string. The function retrieves the parameters requested by the format string from the stack

- **Example: printf(fmt_str,arg_1,...,arg_n).**

- Exploiting format strings
  - If an attacker is able to provide the format string to a format function, a format string vulnerability is present

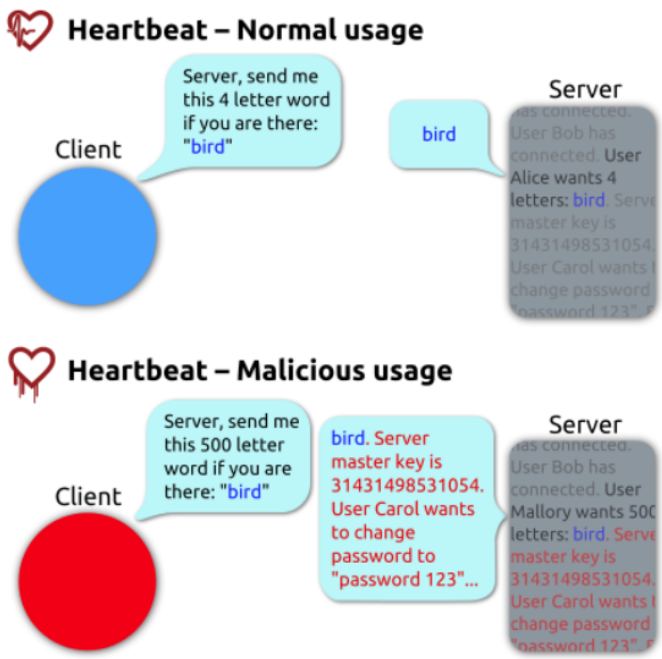
```
int vulnerable_print(char *user) {
  printf(user);
}

int safe_print(char *user){
  printf ("%s", user);
}
```

  - **We can view the stack memory at any location**
    - walk up stack until target pointer found
    - printf (“%08x.%08x.%08x.%08x.%08x|%s|”);
    - A vulnerable program could leak information such as
    - passwords, sessions, or crypto keys
  - **We can write to any memory location**
    - printf(“hello %n”, &temp) – writes ’6’ into temp
    - printf(“hello%08x.%08x.%08x.%08x.%n”)
- More buffer overflow opportunities

- Exception handlers
- Function pointers
- Double free

- TLS Heartbleed



Then, OpenSSL will uncomplainingly copy 65535 bytes from your request packet, even though you didn't send across that many bytes:

```
/* Allocate memory for the response, size is 1 byte
 * message type, plus 2 bytes payload length, plus
 * payload, plus padding
 */
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;

/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);

r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload +
```

That means OpenSSL runs off the end of your data and scoops up whatever else is next to it in memory at the other end of the connection, for a potential data leakage of approximately 64KB each time you send a malformed heartbeat request.

- One of the most common attacks on memory safety

The CWE Top 25

Below is a brief listing of the weaknesses in the 2020 CWE Top 25, including the overall score of each.

Rank	ID	Name	Score
[1]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.82
[2]	CWE-787	Out-of-bounds Write	46.17
[3]	CWE-20	Improper Input Validation	33.47
[4]	CWE-125	Out-of-bounds Read	26.50
[5]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	20.69
[7]	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	19.16
[8]	CWE-416	Use After Free	18.87
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	17.29
[10]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	16.44
[11]	CWE-190	Integer Overflow or Wraparound	15.81
[12]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	13.67
[13]	CWE-476	NULL Pointer Dereference	8.35
[14]	CWE-287	Improper Authentication	8.17
[15]	CWE-434	Unrestricted Upload of File with Dangerous Type	7.38
[16]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.95
[17]	CWE-94	Improper Control of Generation of Code ('Code Injection')	6.53
[18]	CWE-522	Insufficiently Protected Credentials	5.49
[19]	CWE-611	Improper Restriction of XML External Entity Reference	5.33
[20]	CWE-798	Use of Hard-coded Credentials	5.19
[21]	CWE-502	Deserialization of Untrusted Data	4.93
[22]	CWE-269	Improper Privilege Management	4.87
[23]	CWE-400	Uncontrolled Resource Consumption	4.14
[24]	CWE-306	Missing Authentication for Critical Function	3.85
[25]	CWE-862	Missing Authorization	3.77

- ## Lecture 24 - Memory safety defenses
    - ### Key techniques against memory safety attacks
        - 1. Use memory-safe languages - checks on buffer bounds are automated by the compiler
            - **Memory-safe languages are not subject to memory safety vulnerabilities:**
                - Access to memory is well-defined
                - Checks on array bounds and pointer dereferences are automatically included by the compiler
                - Garbage collection takes away from the programmer the error-prone task of managing memory
            - **Plenty of memory-safe languages:** Java, Python, Rust, Go, etc
            - **Whenever possible in new projects use a memory-safe programming language!**
        - 2. Apply safe programming practices - when using non-memory safe languages check all the bounds, and validate user input
            - **Use safe C libraries - Size-bounded analogues of unsafe libc functions**

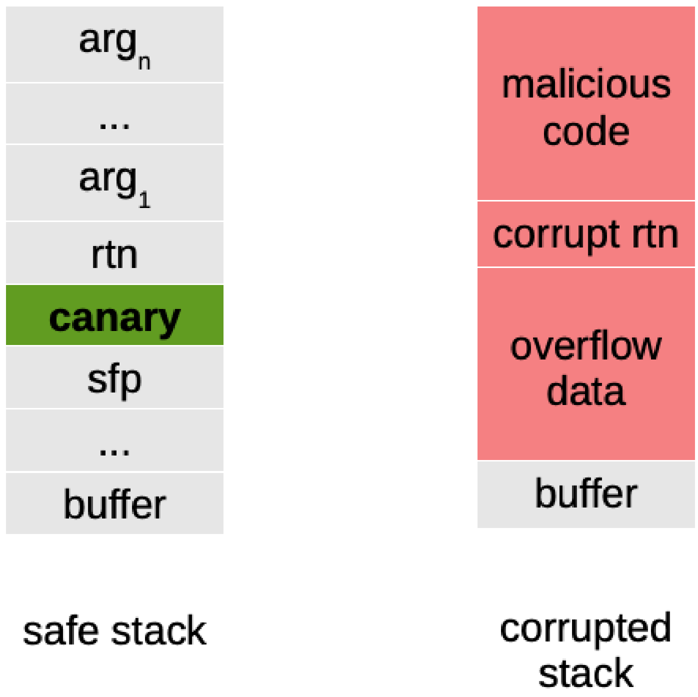
```
size_t strlcpy(char *destination, const char *source, size_t size);
size_t strlcat(char *destination, const char *source, size_t
size);
char *fgets(char *str, int n, FILE *stream);
...
```

            - Check bounds and validate user input

```
#include <stdio.h>
int main(int argc, char *argv[]){
  // Create a buffer on the stack
  char buf[256];
  // Only copy as much of the argument as can fit in the buffer
  strcpy(buf, argv[1]);
  // Print the contents of the buffer
  printf(``%s\n'', buf);
  return 1;
  }
```

```
#include <stdio.h>
int main(int argc, char *argv[]){
  // Create a buffer on the stack
  char buf[256];
  // Only copy as much of the argument as can fit in the buffer
  strlcpy(buf, argv[1], sizeof(buf));
  // Print the contents of the buffer
  printf(``%s\n'', buf);
  return 1;
  }
```

- 3. Code hardening
    - OS and compiler based techniques to defend against BOs
        - 3.1 Stack canaries



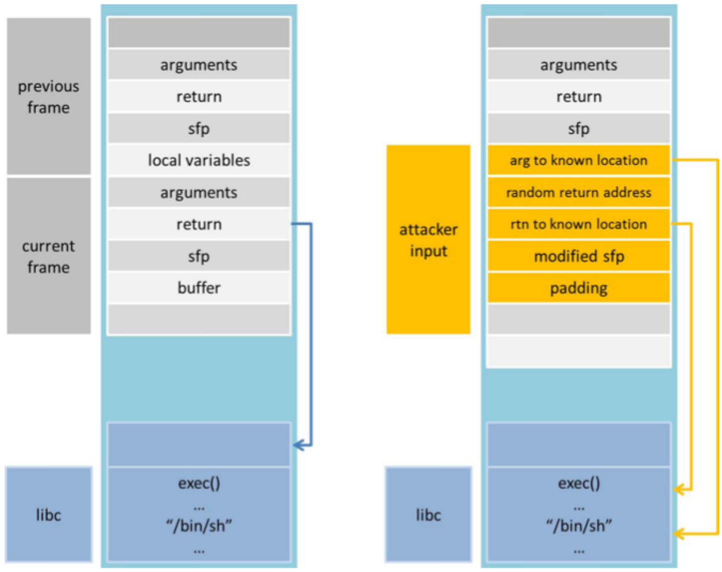
- **Goal:** detect a stack buffer overflow before execution of malicious code
- **Idea:** place trap (the canary)just before the stack return pointer
- The value of the canary needs to be a randomly chosen fresh value for each execution of the program

- To overwrite the return pointer the canary value must also be overwritten
- The canary is checked to make sure it has not changed before returning
- **Limitations**:
    - Stack canaries will detect a BO if
        - The attacker does not learn the value of the canary - this could happen through a buffer overread
        - The attacker cannot jump over the canary - the assumption is that the attacker has to write consecutively memory from buffer to return address
        - The attacker cannot guess the canary value - on 32-bits the attacker might be able to brute force the canary value
        - The buffer overrun occurs on the stack - canaries will not detect heap overruns
- **Stack canaries make attacks harder but not impossible!**

3.2 Data Execution Protection (DEP) / Write XOR execute (W^X)

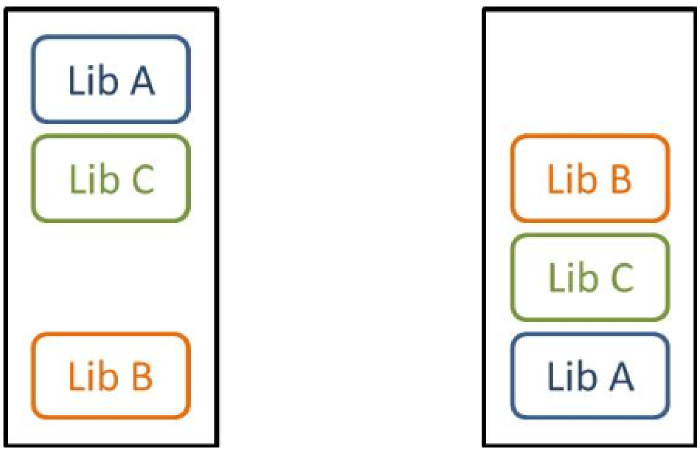
- **Goal:** prevent malicious code from being executed
- **Idea:** Make regions in memory either executable or writable (but not both)
- The stack and heap will be writable but not executable because they only store data
- The **text segment** will only be **executable** and not writable because it only stores code
- Even if the attacker manages to put his malicious code on stack or heap, it will never get executed :-)

- **Limitation of W^X: return-to-libc attacks**:



  - The attacker does not need to inject any code
  - the libc library is linked to most C programs
  - libc provides useful calls for an attacker

- 3.3 Address SpaceLayout Randomisation (ASLR)



- **Goal:** prevent that attacker from predicting where things are in memory
- **Idea:** place standard libraries to random locations in memory
  - $\rightarrow$ for each process, exec() is situated at a different location
  - $\rightarrow$ the attacker cannot directly point to exec()

        - Supported by most operating systems (Linux, Windows, MAC OS, Android, IOS,...)
- But ultimately
    - Hackers have and will develop more complicated ways of exploiting buffer overflows.
    - It all boils down to the programmer.
    - The most important preventive measure is: **safe programming**
    - Whenever a program copies user-supplied input into a buffer **ensure that the program does not copy more data than the buffer can hold**
    - Take away message
        - OSes may have features to reduce the risks of BOs, but the best way to guarantee safety is to remove these vulnerabilities from application code.