C3 Buffer Overflow

1. Background

- Implementation of network protocols often written in C/C++
  - Management of memory left to programmer for performance optimizations
  - Trade-off of **performance over robustness**

Memory Access

- Strings written to / read from memory which are accessed via pointers
- "Unsafe" write: given a pointer and string, write to memory starting at the address pointed to until the end of the string (NUL character `\0`)
  - No warning generated when too many characters are written
- "Safe" write: given a pointer, string and **bound**, write to memory until end of string or bound

1.1. Variables and Buffers

- Program stores data in variables
- A region of memory (buffer) is allocated to store the **value** assigned to the variable
- Possible writing to wrong location / writing more data than buffer can hold

Buffer Overflow (bof)

- If **value** assigned to variable exceeds the size of allocated buffer, memory not allocated to this variable is overwritten
  - If overwritten belongs to allocated memory of another variable, the value of that variable is changed
- Different kinds of buffer overflow
  - Stack-based
  - Heap-based
- Data written to a buffer is written upwards (towards higher addresses) from address of buffer
- **Unintentional bof** crashes software (segfaults) and have been a focus for reliability testing
- **Intentional bof** is a concern if an attacker can modify security relevant data
- Attractive targets include **return addresses** which specify next piece of code to be executed and **security settings**
- Combat using **Type-safe** languages to guarantee error-free memory management

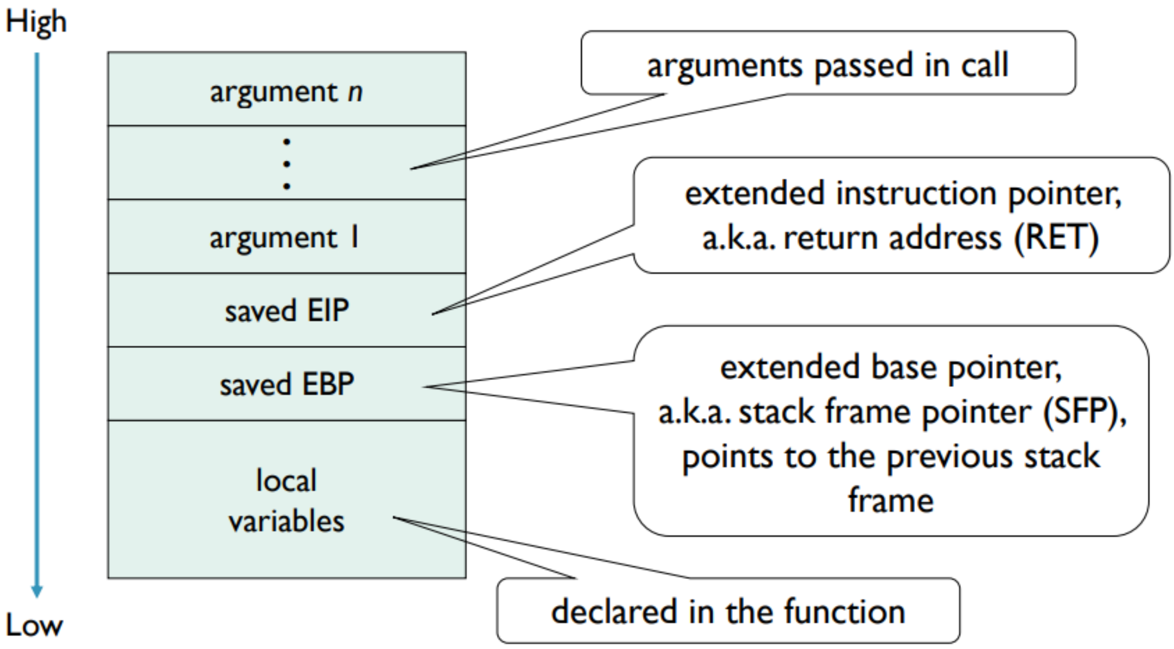
1.2. Call Stack

Function Calls

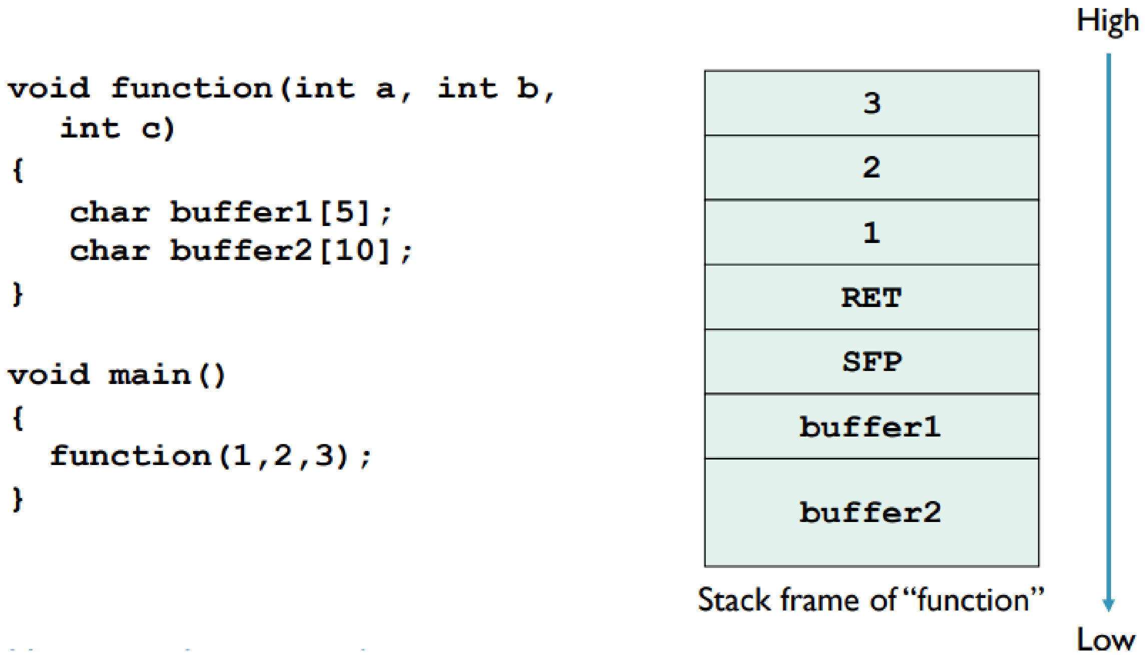
- Structured programs use functions (subroutines, methods) which may call further functions
- When runtime env executing a program encounters a function call, it collects data needed (arguments etc) and starts executing the function in a frame (space allocated on **call stack** for function execution, away from driver code)
  - Runtime env returns to caller when the execution of the function is completed

Stack and Heap

- Stack: Top down (towards lower addresses) allocated memory; relatively easy to figure out where a given buffer will be placed on the stack
- Heap: dynamically allocated memory (upwards), difficult to guess where a given buffer will be taken from the heap; depends on memory allocation scheme
- **Stack frames** contain **return address, local variables, function arguments** as well as user data and control data
- When call returns from function execution, execution continues at the return address specified



High
argument *n*
argument 1
saved EIP
saved EBP
local variables
Low
arguments passed in call
extended instruction pointer, a.k.a. return address (RET)
extended base pointer, a.k.a. stack frame pointer (SFP), points to the previous stack frame
declared in the function



```
void function(int a, int b,
   int c)
{
    char buffer1[5];
    char buffer2[10];
}

void main()
{
  function(1,2,3);
}
```

Stack frame of “function”

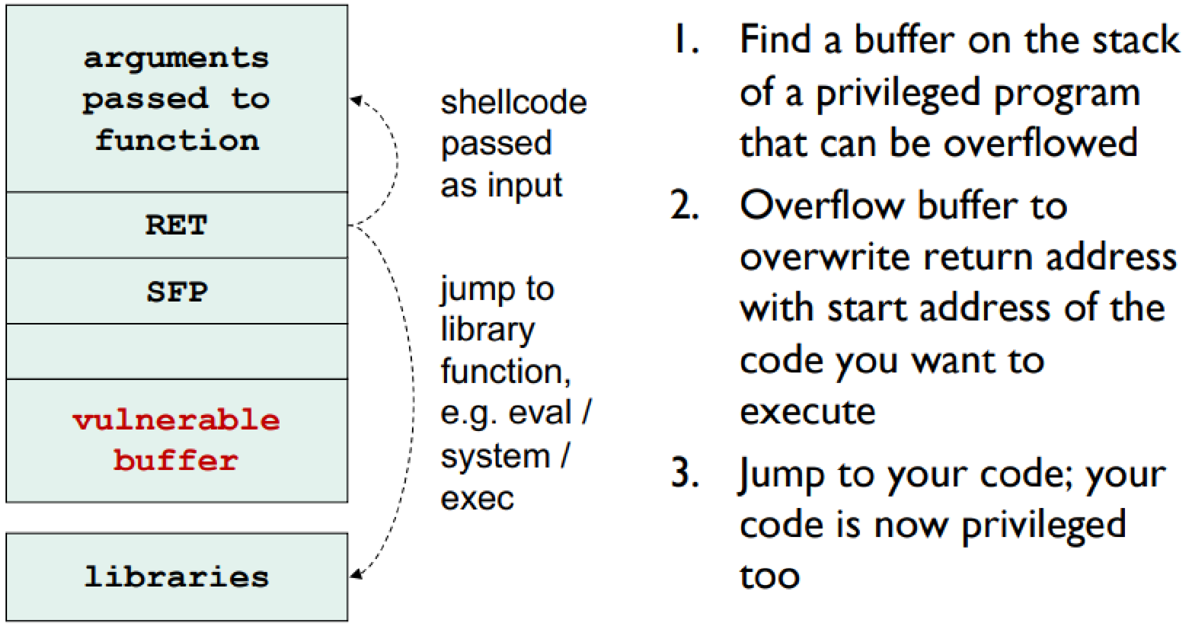
- in 64 bit architecture, the parameters to any function is passed using registers rather than by pushing it into stack. For linux, parameter passing is done in this order: RDI, RSI, RDX, RCX, R8, R9, remaining from the stack
- first six values leaked will be values from registers and not the top value off the memory stack

2. Buffer Overflow Attacks

2.1 Pattern

1. Find function containing unsafe write of **user defined input** to local variable
2. Attacker supplies specially crafted input into function
3. Input overflows buffer allocated to local variable until it **overwrites return address**, making return address point to attacker's code (shellcode) instead of back to calling program

2.2 Stack-Based Bof


arguments passed to function
RET
SFP
vulnerable buffer
libraries
shellcode passed as input
jump to library function, e.g. eval / system / exec

1. Find a buffer on the stack of a privileged program that can be overflowed
2. Overflow buffer to overwrite return address with start address of the code you want to execute
3. Jump to your code; your code is now privileged too

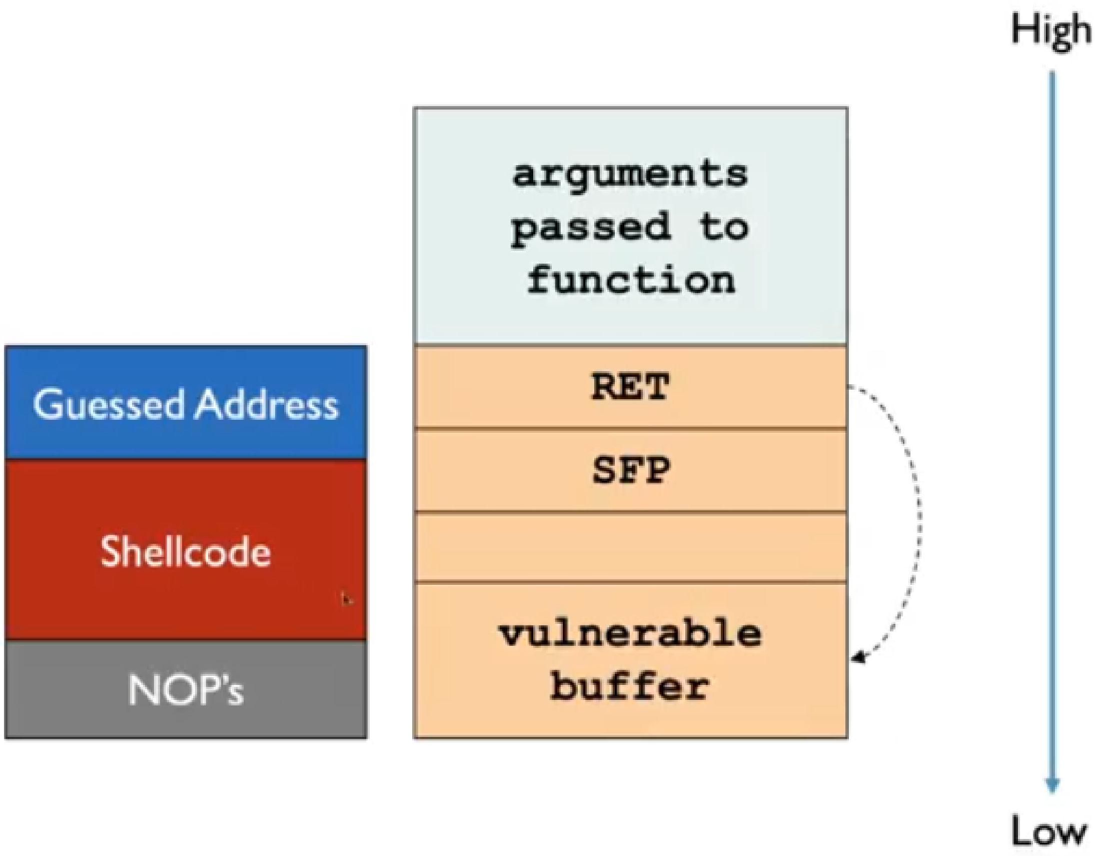
2.3 Shell Code

- System is not corrupted yet when the attack starts
- `argv[]` method: put shellcode on stack as part of malign input
  - Attacker has to guess distance between return address and address of argument containing shellcode
- `return-to-libc` method: jump to `eval` library function that will execute commands provided as user input
- Return oriented programming (ROP): jump to code segment in some existing executable

Detour (Relative addressing)

- Attacker only has to guess relative address of shellcode

- actual address = base + offset (array index) x size (size of array index)



- Input on the left is input as buffer, as seen, SFP (stack frame pointer) and RET (return address is overwritten), in particular, RET is overwritten with guessed addresses which point to NOP (landing sled) for shellcode to execute
  - **shellcode is input into the buffer**

ret2libc

- Return address changed so that execution returns a library function which expects its arguments on stack
- Attacker puts arguments to library function in a ghost stack frame
- **Defence**: library functions that take arguments only from CPU registers

3. Defences

Countermeasures

- "Location" classification
  - Programming language
  - Libraries
  - Compiler
  - Hardware
- Security strategies classification

    - Prevention
    - Detection
    - Mitigation
- Defended in steps:
    1. Check input arguments for variables / arrays so input size fit memory allocated
    2. Protect return address
    3. No executable inputs
    4. Make guessing of potential shellcode / system library location difficult

3.1. Safe Functions

- Replace unsafe string functions by using functions where the number of bytes to be handled are specified
    - `strncpy`, `snprintf`, `fgets`
- Need to get arithmetic right and know the correct maximal size of data structures

`strsafe.h`

- Header used for undefining unsafe functions (`#define strcpy unsafe_strcpy`)
    - Raises error during compile time when code contains `strcpy` since `strcpy` is defined to something else
- No overflow of destination buffer
- Guarantees null-termination of buffers

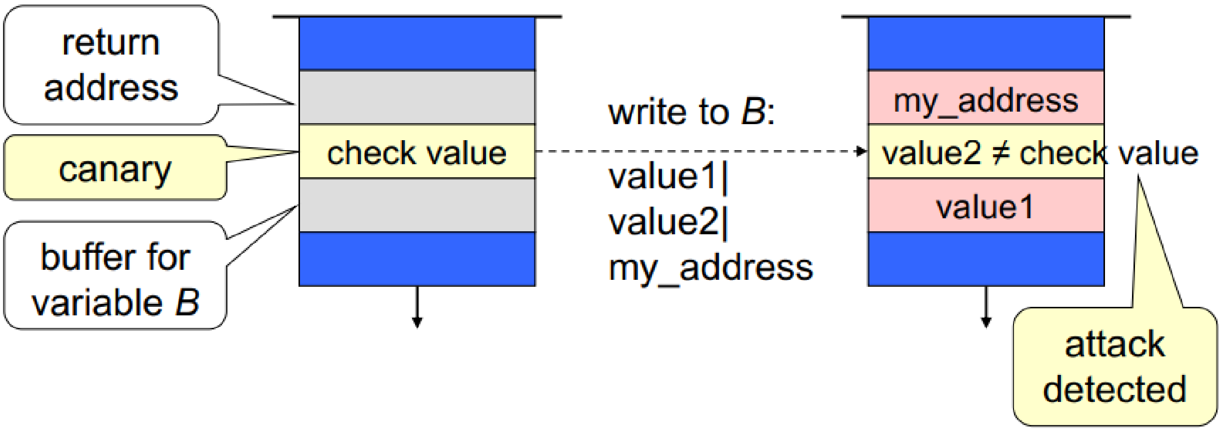
Type Safe Languages

- Use programming languages where bof cannot happen
- Guaranteed by static checks and runtime checks
    - Array bound checking
    - Garbage collection

3.2. Canaries

- Compiler places a check value (canary) in memory location just below return address, checks canary is not changed before returning
- Stack guard: random canaries placed in stack

- Source code has to be recompiled to insert placing and checking of canaries

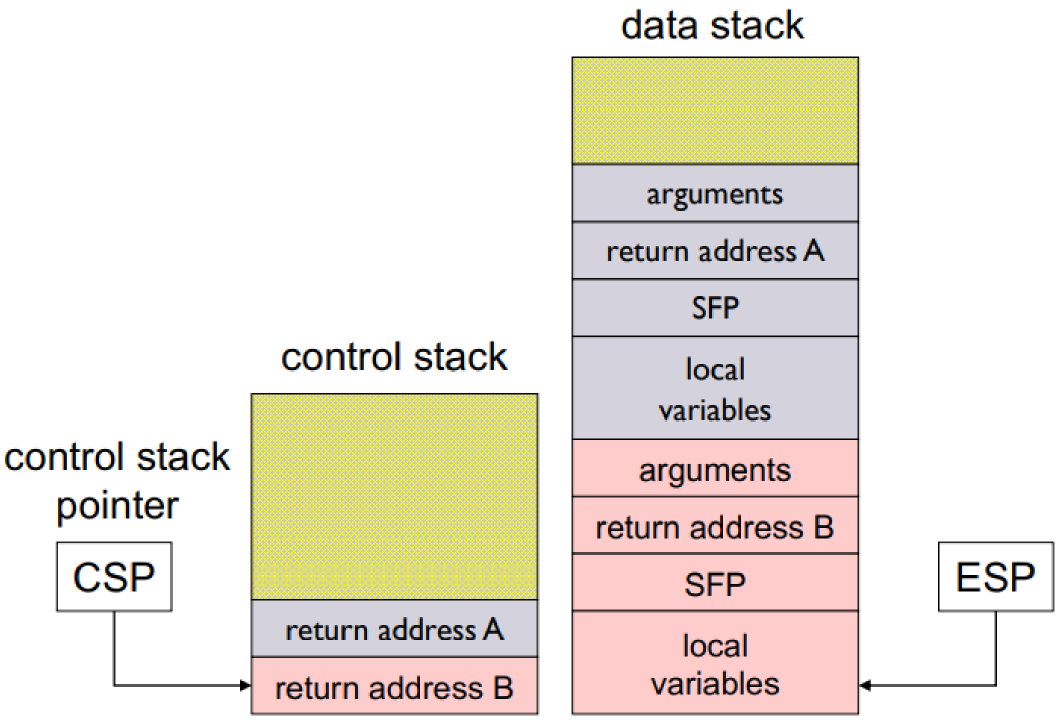


Requirements

- Check values must not be predictable
- Reference values must be integrity protected (written to protected memory / calculated using secret cryptography keys which must be protected)

3.3. Split control and data stack

- Stack-based bof attacks overwrite return addresses
  - Would be blocked if the return address is not taken from stack but from a location the user cannot modify
- Control stack: memory area separate from data stack which the user cannot modify



3.4. Address Space Layout Randomization

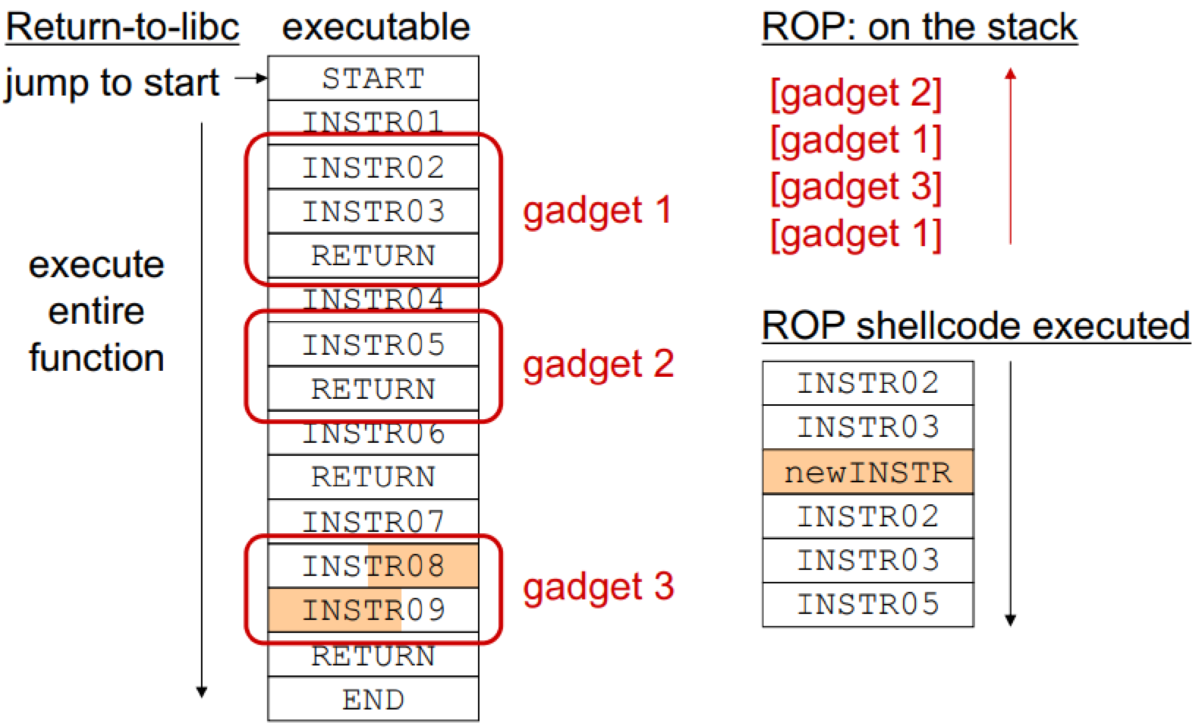
- Stack bof attacks have to know position of target buffer in relation to return address and the approximate location of attack code
- By changing memory allocation randomly, this reduces the chances of attacker guessing locations correctly

3.5. Data Execution Prevention (DEP)

- Marks memory location a process has written to as non-executable
- W xor X protection
  - Memory can be writeable / executable but not both, this makes shellcode placed by attacker unable to execute after being written into memory
- Cannot be applied to components creating executable code

4. Return-oriented Programming

- Attacker can use existing executables (from system libraries) to bypass DEP, but is limited by this factor
- Attacker uses short code segments (gadgets) in executables that end with a return command
  - When misaligned memory access is permitted the attacker may find byte sequences in the middle of a word that constitute valid machine instructions
  - Gadgets serve as building blocks for writing shellcode, enough gadgets can make shellcodes with arbitrary functionality
- Stack based bof puts gadget addresses on stack, program will execute gadget after gadget, using call stack as a trampoline



- More powerful than ret2libc attacks