PWN fundamentals

(this powerpoint is focused on the stack and its vulnerabilities)

By Jarrett Lane

What are PWN challenges?

- PWN, also known as binary exploitation, is about finding vulnerabilities in executables and low level code. It is similar to reverse engineering, but with these challenges the focus is more about how to exploit the code rather than what the code does.
- For these challenges it is useful to have a good understanding of low level machine concepts, binary executables (.elf (linux) or .exe files), the C/C++ language, good reversing skills, and a good understanding of how a program executes.

Common approaches to PWN

- Gather information: See what type of binary, operating system, and architecture is used. This is very important to know because of things like endian format and more.
- Analyze protections: It is important to know which kind of security mechanisms are already in place, like stack canaries, PIE, etc.
- Static/Dynamic analysis: When reversing the actual binary, there are two ways of doing so, statically & dynamically
- Identify: look for noticeable vulnerabilities, or do some testing to see if you can make the program do something it shouldn't, once you do this you can develop an exploit to this vulnerability

Accessing challenges

- Usually these types of challenges can be accessed via a netcat connection, they are remote challenges because unlike reverse engineering, you are attacking a running machine on the other end. You are given the binary to test locally on your computer, and once you figure out the vulnerability you use it on the remote system where the actual flag is stored.

What is netcat?

- Netcat is a versatile networking tool that can set up connections & a lot more, its open source and can be used on any system, in this case it lets you connect to the vulnerable executable running on the CTF organizers servers to get the flag
- Example: nc dicec.tf 32030 <--- When you run this, it connects to the domain dicec.tf on port 32030 and the challenge is accessed

Good tools for PWN

- Ghidra & IDA free: reverse engineer the binaries (static analysis)
- GDB (or pwn tools GDB extension): debug the binaries (dynamic analysis)
- PWN tools: python library for pwn activities
- Checksec: check which protections are being used in the binary
- There are alot more tools, but I think these tools are the main ones used

Static vs Dynamic analysis

- Static analysis: analyze the code without actually running it
- Dynamic analysis: analyze the code as it is running
- Each analysis has its uses, it depends on the type of challenge

Common vulnerabilities in PWN

- Buffer overflow: user input takes more space than it is allocated, without protections this input can overwrite other parts of memory, which can be used maliciously to run unexpected code and more
- Format string: when functions like printf() misinterpret user input, the use of format strings can be used maliciously to read unauthorized data and more

Low level basics

- Low level means closer to the machine/hardware. This means less human readable and more relevant to how the computer itself is working and interpreting data
- In these next few slides, I want to introduce low level topics such as the stack, memory basics, and the execution of programs, which are important to understanding PWN

The stack

- Do you remember when you were a kid, you probably remember this toy. The little blue ring was the last ring to be put on, so that means it will be the first taken off. This is how a stack works.
- This same concept is used in computers too, and it can be applied to how programs run

Memory

- Memory is where data is stored. It is a fundamental part of a computer.
- Memory is organized by addresses, and data can be found in memory by knowing its address.
- Memory can be classified as volatile or non-volatile

Volatile vs non-volatile

- Volatile memory (RAM): This memory is also known as temporary memory, it is memory that can be quickly accessed and changed, and it is the memory that programs occupy when they are running, having lots of volatile memory allows the computer to run lots of programs (Although you better have a good CPU)
- Non-Volatile memory: This memory also known as storage, stores data, even when the computer is shut off. Usually it is slower to access than RAM but it can hold lots of information, this is good for long term storage
- For PWN challenges we will be discussing volatile memory because we are talking about running programs

Pointers

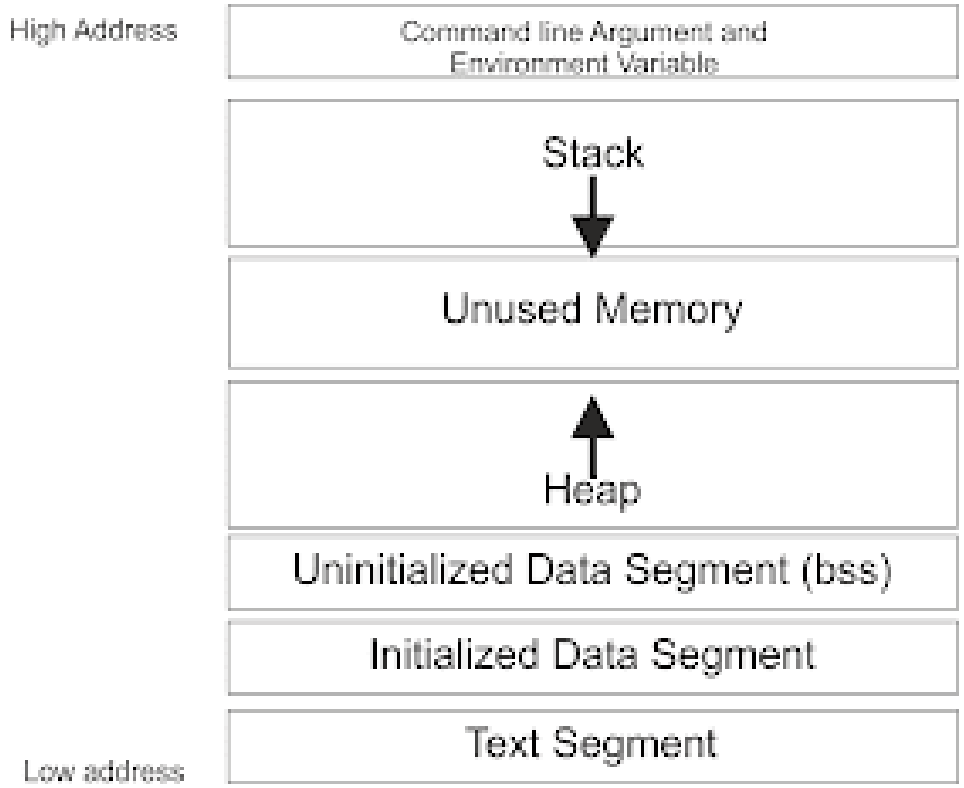
- A pointer is a reference to another memory address, which means if I am storing a pointer in my program, I am storing the address of a specific place in memory.
- Pointers are used for manipulating memory, using data structures, operating on arrays, and a lot more
- In C, a pointer looks like: int *Number, this means our pointer which we are calling "Number" is storing a memory address that stores an integer

What are binary executables?

- When you write code for a program, the computer is not reading your code, it is reading binary instructions that do what your code does.
- These binary instructions are called machine code, and it is how the computer understands how to run a program, to convert your written code to machine code is called compiling, so compiled code is code that is meant to be understood by the machine
- Binary executables are compiled programs, which means the source human readable code has been translated to code that the machine understands

Memory layout of running programs

- When you run binary executables, RAM/Volatile memory is needed to store program data when the program runs. This diagram is how the memory is organized. As a program runs, the stack and the heap grow and shrink, but for this meeting we will focus on the stack



High vs Low addresses

- Addresses are usually represented as hexadecimal values ranging from 0 to F, we say that 0 is low and F is high, so when we say that the stack grows down in memory, we mean that when we add elements to a stack, the newer elements are at memory addresses with lower hex values

What is a register

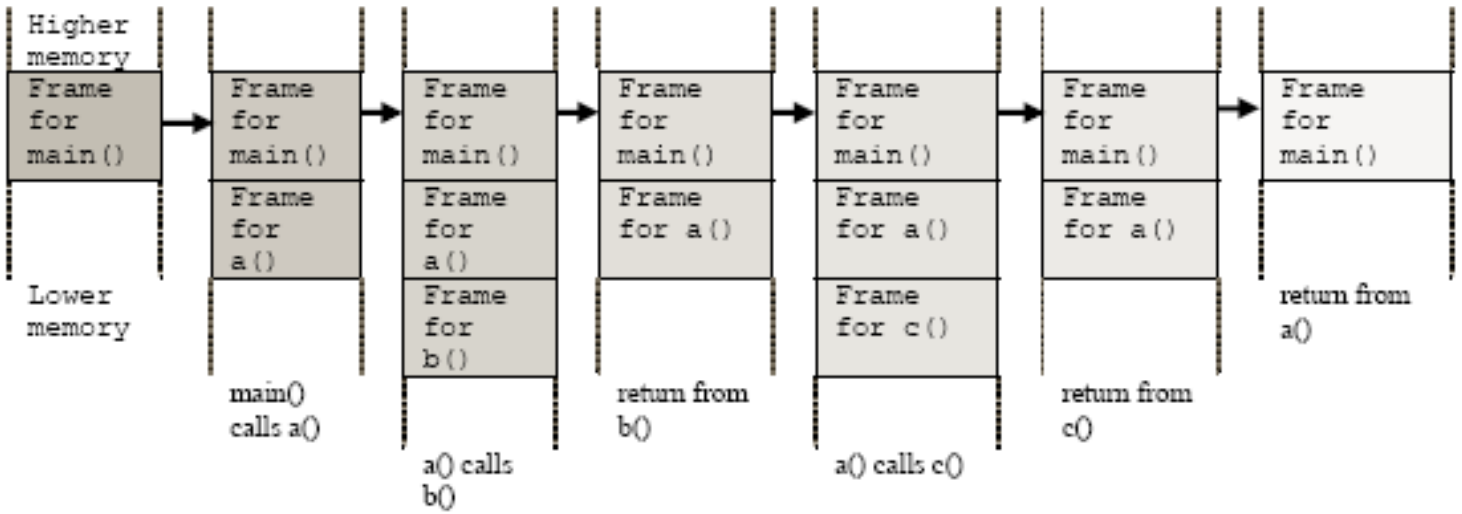
- The CPU (central processing unit) takes instructions and executes them, it is a fundamental part of any computer
- When a program is running, the CPU and RAM need communicate with each other to run the program
- the CPU handles the actual execution and writes the results to the volatile memory
- Just like how a program used volatile memory, the cpu executing the instructions also has some memory that it can access within itself called registers, these are needed to operate on values and maintain organization of the running program with pointers

The stack during execution

- When you execute a program, the computer needs to have an organized way to know which routine is running, which routine to return to, and which data is relevant to a routine, this is where the stack comes into play
- The stack, just like that toy I showed you earlier, will take in routines/functions as they are called and pop (or remove) those functions when they return/finish, with the most recent addition being removed first
- The stack is crucial to how a program runs, and if an attacker can modify it, very dangerous things can happen

The stack during execution simple example

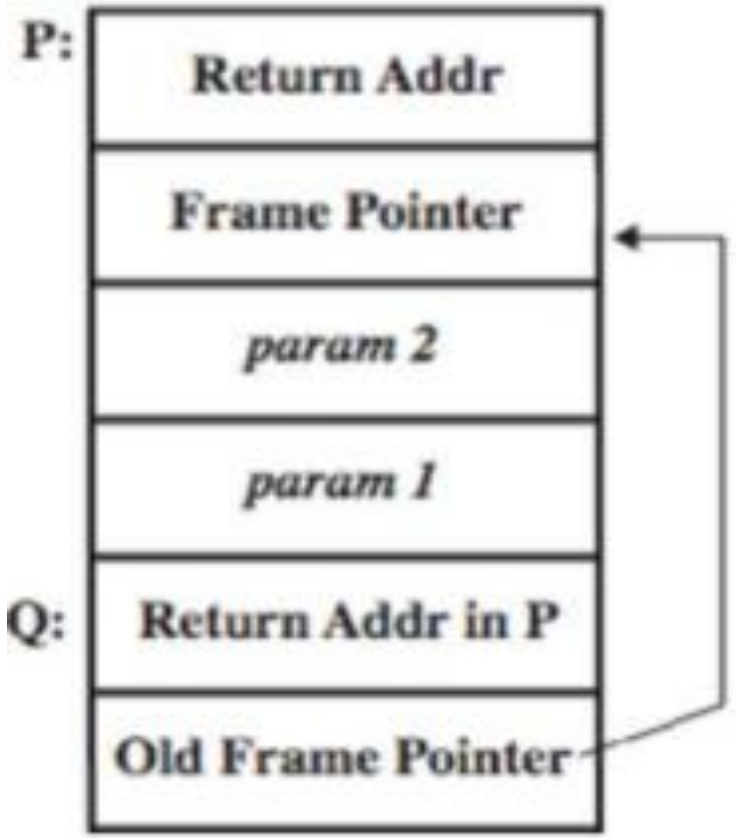
```c
#include <stdio.h>

int a();
int b();
int c();

int a()
{
  b();
  c();
  return 0;
}

int b()
{ return 0; }

int c()
{ return 0; }

int main()
{
  a();
  return 0;
}
```



Stack frame and function call.

Stack frame

- A stack frame is a part of the stack that has all of the data relevant to a certain function call, usually containing a return address to the last function that was called, a frame pointer that separates the frames and defines the start of the current frame, Usually this pointer is stored in a register, but if another function is called, the callers frame address is saved in the called function, so when the called function finishes, it can read that old address from the stack and correctly return to the right frame.



%rsp %rbp

- Remember registers? They are important to the CPU, and two of them are reserved for handling the stack
- %rsp points to (holds the address of) the top of the stack
- %rbp points to the stack frame, think "which function am I dealing with right now?"
- These allow the CPU to stay organized and execute the order of the stack properly

Getting into the exploits

- These concepts may seem like a lot at first, but once we get into the exploits it will make more sense and it will all come together
- As long as you have a general understanding of how this all actually works you will be able to understand the exploits
- There are many exploits that can occur with binary executables, but we will focus on buffer overflows and format string vulnerabilities
- If you have any confusion about the past slides, please ask questions before we continue

This is me when I see unsafe input functions or printf statements that don’t specify a format

Buffer Overflow

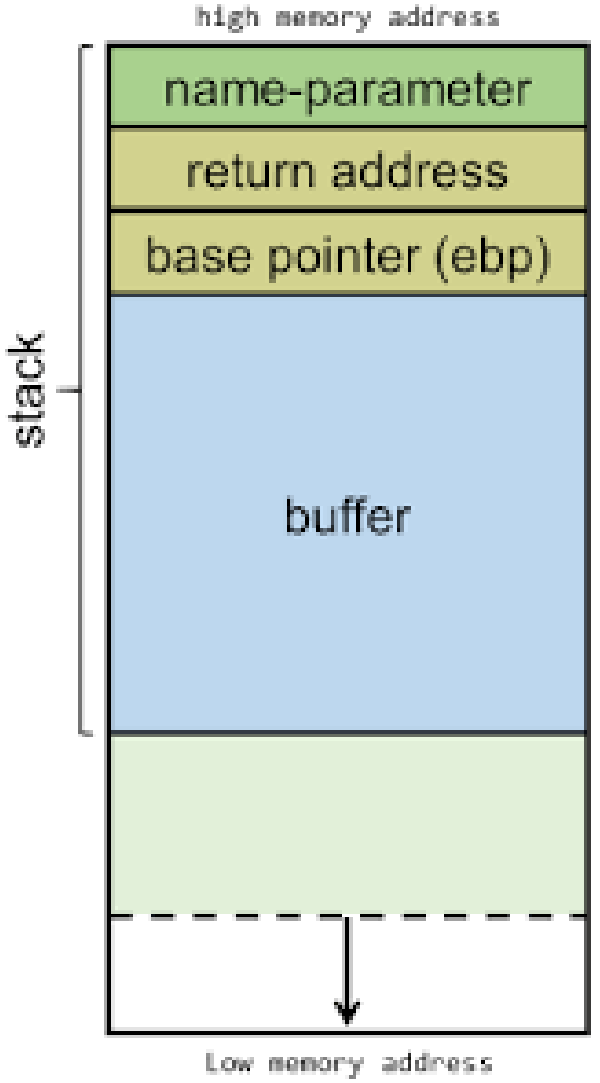
- Remember how I said the stack is crucial to how the program runs, what If you could somehow overwrite the contents of the stack and make it do what you want or change what you want, this is what a buffer overflow vulnerability does
- When a function takes input, the input is held in the stack, in the stack there is allocated memory reserved for this input called the buffer
- Unlike other languages that are more high level, in the C language, the size of the buffer (the amount of space that is allocated for holding the input) must be known before it is compiled

Buffer issues

- Because of the way C handles buffers, problems can arise
- What if the amount of space allocated for the buffer is not enough? Depending on how safe the program is, two things can happen
- If the program is safe, it will impose the limit of data that can be held and only a permitted amount of data will be passed to the stack
- If the program is unsafe, the data will extend beyond the buffer and start overwriting adjacent data on the stack , this can have various consequences
- When coding in C, it is best practice to make sure you handle input safely so it doesn't overwrite anything, and also make sure to give a reasonable amount of allocated space for inputted data, a reasonable amount depends on the use of the buffer

Overwriting data

- When you write too much to a buffer, it will start to overwrite the values in the higher addresses, lets say for example the input was handled unsafely and it overwrites the return address, what will happen is when the stack reaches it, it wont see a proper address and will crash because it didn't know where to return to in order to keep executing
- What an attacker can do is calculate the right amount of data needed to overwrite the stack until the return address is reached, and then change the return address to another function, then when the payload is input, a function of the attackers choice will be run



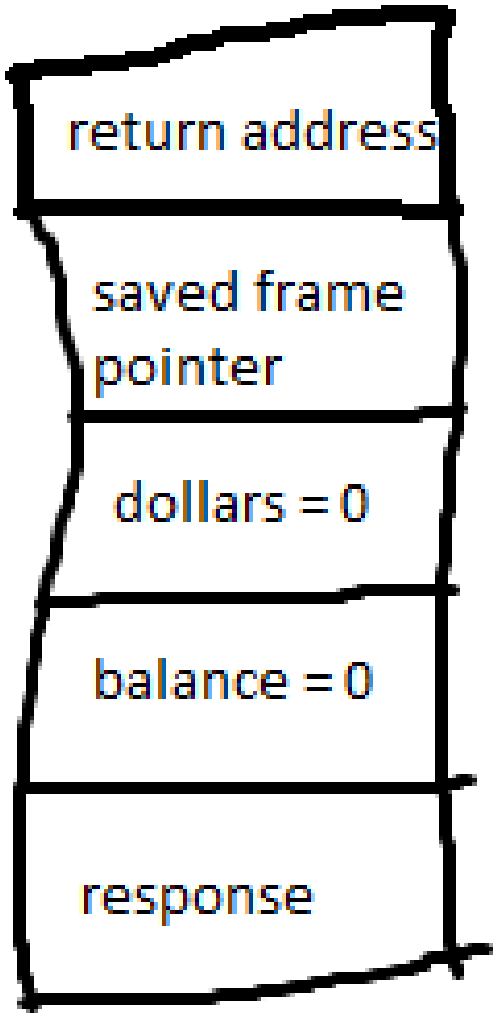
An example of overwriting data

- I wrote this banking application, can you get more than 0 dollars?

```c
#include <stdio.h>

int main()
{
    int dollars = 0;
    printf("Welcome to the bank, you have %d dollars, input an 8 digit code to check your balance again: ", dollars);

    int balance = 0;
    char response [8];
    gets(response);
    dollars = balance;
    printf("You have %d dollars, have a good day", dollars);

    return 0;
}
```

An example of overwriting data

- When this code runs, this is what the stack looks like, we know that we allocated 8 characters to the buffer/response, if we make up any 8 digit code and then add a numeric value, that could overwrite the balance, which gives us a new balance
- I'm going to post this code in the CTF home made channel for you to try



How to avoid this exploit

- Avoid using functions that are known for handling input unsafely
- Randomize addresses to make it unpredictable for the attacker to know where to overwrite (ASLR)
- Sanitize the input for unsafe data or too many characters
- Try to use languages that can handle buffer sizes and is safer with memory

Unsafe Function	Risk	Recommended Alternative
gets()	Unbounded input	fgets()
strcpy()	No length checking	strncpy()
scanf()	Buffer overflow	sscanf() with size limit

Buffer overflows in the real world

- There are many different types of buffer overflow exploits, some allow you to execute your own code, some involve evading protections like stack canaries, but for tonight I want to focus on a simple example
- Despite the fact that many high level languages are starting to be used more, buffer overflows are still a prevalent vulnerability in legacy systems and even in new systems
- It is important that programmers are educated about this vulnerability and plan for security before development

Format strings

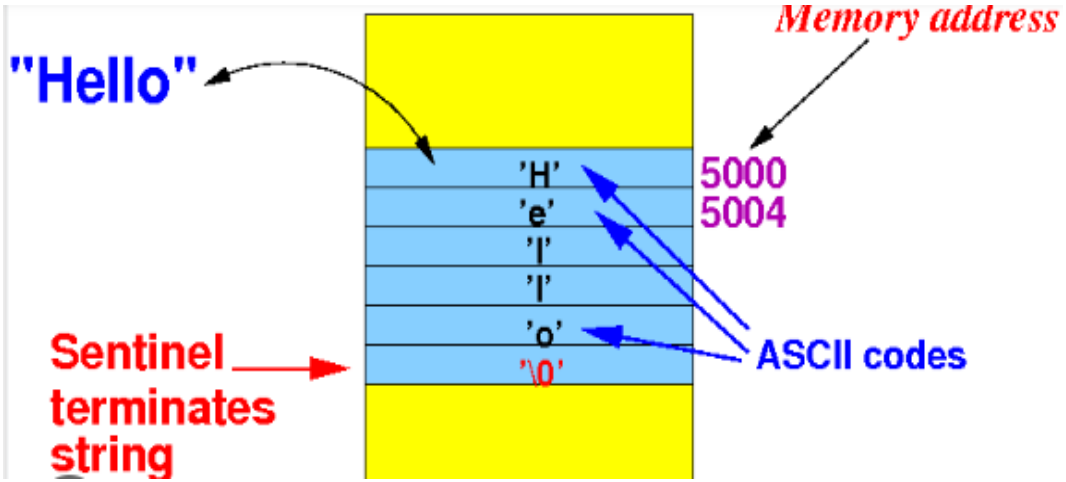
- In C/C++, format strings are used when a string has values that will be interpreted and formatted into the string then output, they are used with functions like printf that handle them
- These strings contain format specifiers which are indicators for how to format variables that are written into the string
- An example: int num = 1, printf("This is an integer: %d", num), the %d specifies an integer value, so when printf is evaluated it will output "This is an integer: 1"

Format specifiers

- When using format strings, common format specifiers are:
- %d: integer
- %x: hex value
- %f: float value
- %s: string pointer
- %n: pointer that tells how many bytes have been written

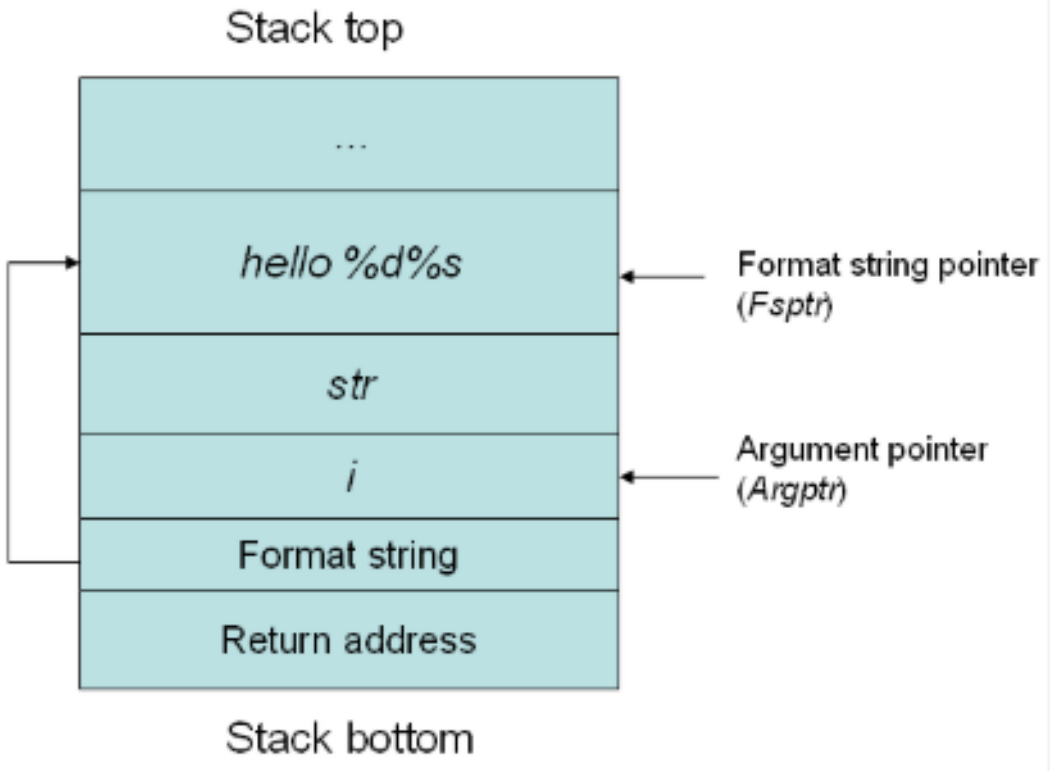
A note about strings in C

- In C strings don't have their own type like other languages, they are represented as character arrays instead
- Pointers are used when handling arrays, the name of the string is a pointer value in memory, and when you go through characters, you are going from an offset from the original pointer that marks the start of an array
- An example of a string is: char string [] = "This is a string in C"



Printf() and the stack

- When printf() runs, it takes in the format string and goes through each character, when it encounters a format specifier it checks the arguments on the stack and resolves them in the string in the specified format



Format string vulnerability

- In some cases, programmers use printf(buffer) instead of printf("%s", buffer), if the buffer is something the user inputs, problems can arise
- If printf() is used without specifying the a format and an angument, if an attacker inputs %x or %s, it will start reading values from the stack because no arguments were ever specified
- For example if an attacker inputs "%x%x%x" it will start printing the contents of the stack in hex

Avoiding this vulnerability

- One way this can be mitigated is by making sure the number of format specifiers matches the number of arguments used, so if an attacker tries pulling off "%x%x%x", there would be three format specifiers but no arguments so that can be caught
- Using the printf() statement like this: printf("%s", string) is much safer than printf(string) because everything is specified properly
- Input validation is also a good way to make sure input is secure

Format strings today

- Many programmers who are unaware of this vulnerability and legacy systems still being used keep this vulnerability relevant, like buffer overflow
- Format string vulnerabilities go deeper than just reading the stack, RCE and much more is possible too

References

- https://wiki.studsec.nl/books/ctf-guides/page/pwn/export/pdf
- https://www.trentonsystems.com/en-us/resource-hub/blog/volatile-vs-nonvolatile-memory#:~:text=At%20a%20high%20level%2C%20the,after%20the%20system%20shuts%20off
- https://knavite.blogspot.com/2014/07/linux-memory-layout-test-through-c.html
- https://www.installsetupconfig.com/win32programming/processtoolhelpapis12_1.html
- https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/
- https://labex.io/tutorials/c-how-to-replace-unsafe-input-functions-422202

More references

- https://www.timusnetworks.com/what-is-buffer-overflow-security-vulnerabilities-and-prevention-methods/
- https://hackinglab.cz/en/blog/format-string-vulnerability/
- https://www.cs.emory.edu/~cheung/Courses/255/Syllabus/2-C-adv-data/string.html
- https://axcheron.github.io/exploit-101-format-strings/