Software Security

Software Vulnerabilities & Attacks

- **Return-to-libc attack**
- **Race Condition vulnerability and attack**
- **Dirty COW (**Dirty copy-on-write)
- **Format String vulnerability and attack**
- **Heartbleed vulnerability and attack**
- **Shellshock vulnerability and attack**

Top 10 Secure Coding Practices by CERT

1. **Validate input**.
2. **Notice Compiler warnings.**
3. **Architect and design for security policies.**
4. **Keep it simple.**
5. **Default to deny.**
6. **Adhere to the principle of least privilege.**
7. **Sanitize data sent to other systems.**
8. **Practice Defense in depth(DiD model).**
9. **Use effective quality-assurance techniques.**
10. **Adopt a secure coding standard.**

Code : Robustness

- When writing code, it is important to ensure that a **program runs correctly and continues to run** no matter what actions a user takes.
- This is done through planning for all possibilities and thinking about what a user may do that the program does not expect.

Code : Robustness

Designing robust programs encompasses three areas:

- protection against **unexpected user** inputs or actions, such as a user entering a letter where a number was expected
- confirming that users on a computer system are **who they say they are**
- minimising or **removing bugs**

Return-to-libc attack

TCS 591 : Unit 2

“Returning to libc”

“Returning to libc” is a method of exploiting a buffer overflow on a system that **has a non-executable stack**, it is very similar to a standard buffer overflow, in that the return address is changed to point at a **new location that we can control**.

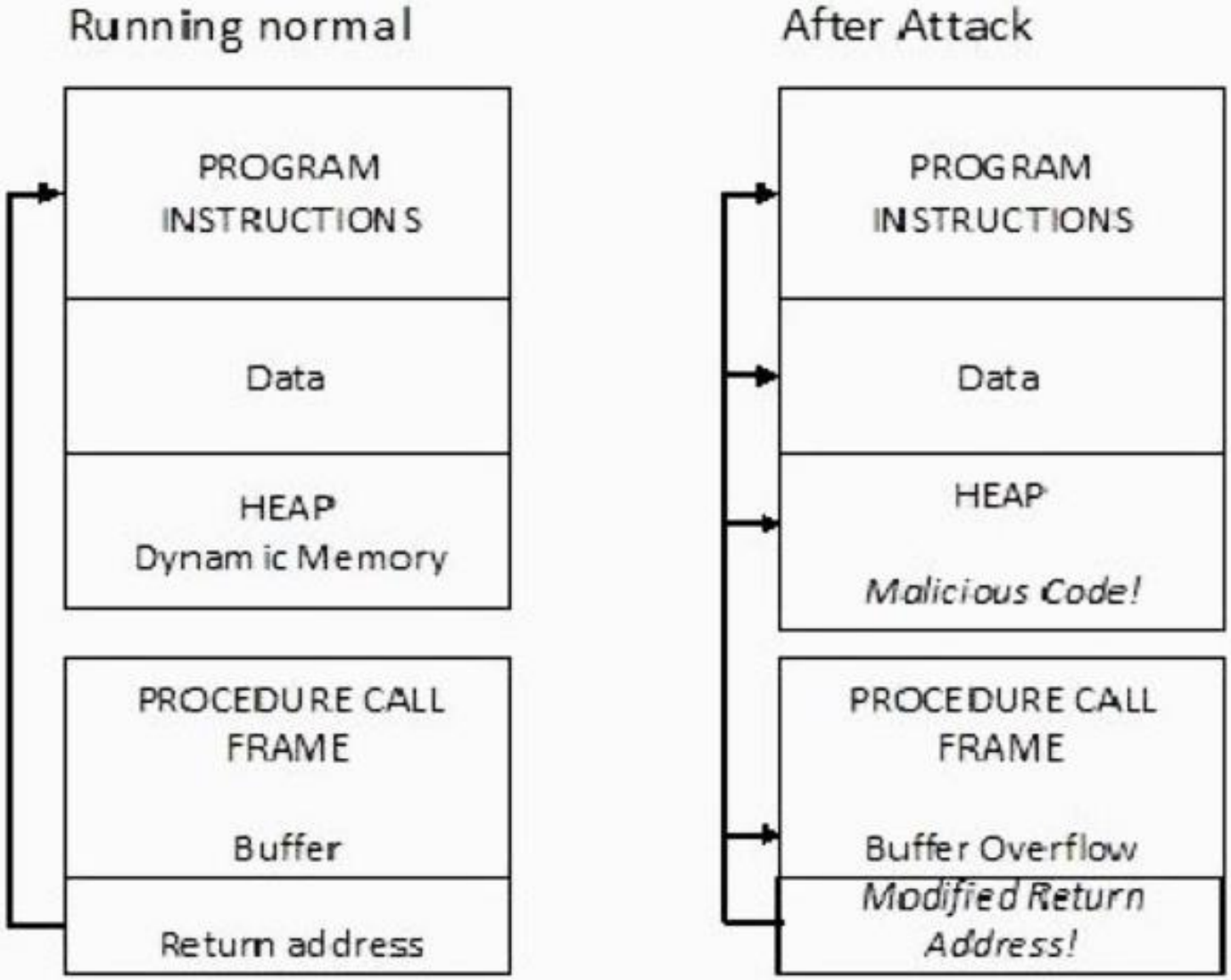
However since no executable code is allowed on the stack we can't just tag in shellcode.

This is the reason we use the return into libc trick and utilize a **function provided by the library**. We still overwrite the return address with one of a function in libc, pass it the correct arguments and have that execute for us.

Since these functions do not reside on the stack, we can bypass the stack protection and execute code.

The C standard library, or libc

- Most programs use common external libraries, and first among them, **the C standard library, or libc**. This library defines such basic functions as “memcmp”, “memcpy” and “printf”, the **building blocks** from which all other code is made.
- If we can overwrite return addresses to point elsewhere in the code, and all code includes these building blocks, then we can overwrite the return address



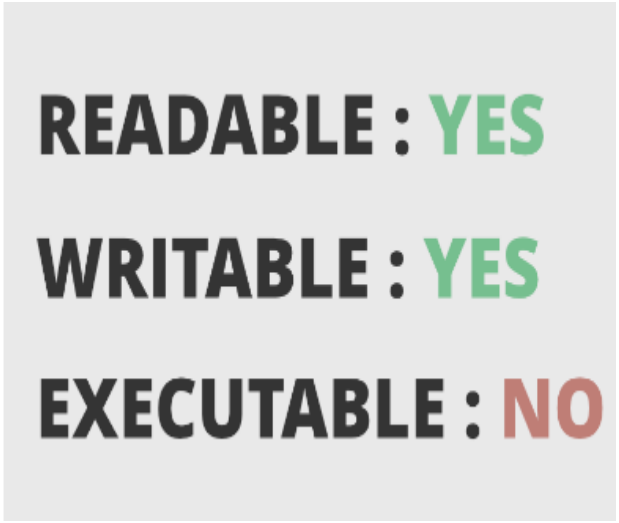
Attacker plants code that overflows buffer and corrupts the return address. Instead of returning to the appropriate calling procedure, the modified return address returns control to malicious code, located elsewhere in process memory.

What is a Shell code ?

- Shellcode is a special type of code injected remotely which hackers use to exploit a variety of software vulnerabilities. It is so named because it typically spawns a **command shell** from which attackers can take control of the affected system.
- **Shellcode is a set of instructions that executes a command in software to take control of or exploit a compromised machine**

Solution : NX bit

- This is often called the **NX bit, for no-execute**, so we can map the stack, that memory region where buffers and **return addresses** are stored, as non-executable.
- Now, even if an attacker successfully overflows a buffer, overwrites the return address, and diverts the program execution to this buffer, the hardware will refuse to execute this buffer as code, and abort the program.



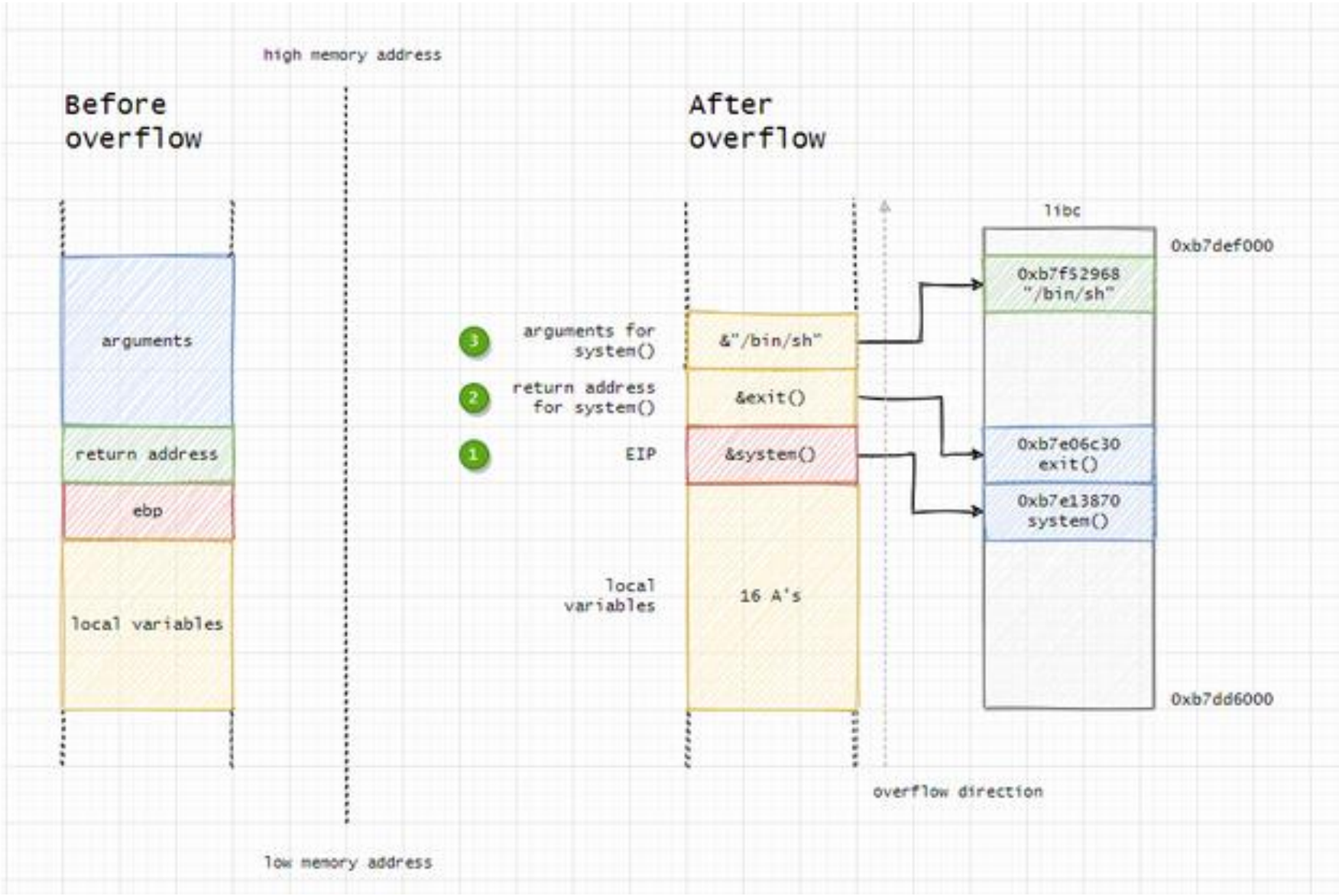
Return to Libc to bypass NX

- Return to Libc is a common and easy technique to **bypass NX**
- No shellcode on the stack(as NX is non executable)
- We will need to find few offsets in libc and redirect execution to libc
- ASLR must be turned off . So , the base address of libc will not change

Address space layout randomization (ASLR) is **a technique that is used to increase the difficulty of performing a buffer overflow attack that requires the attacker to know the location of an executable in memory**.

ret-to-libc

- At a high level, ret-to-libc technique is similar to the regular stack overflow attack, but with **one key difference** - instead of **overwriting the return address** of the vulnerable function with address of the **shellcode** when exploiting a regular stack-based overflow with no stack protection, in ret-to-libc case, the return address is **overwritten with a memory address that points to the function system(const char *command) that lives in the libc library**, so that when the overflowed function returns, the vulnerable program is forced to jump to the system() function and execute the shell command that was passed to the system() function as the *command argument as part of the supplied shellcode.


high memory address
Before overflow
arguments
return address
ebp
local variables
low memory address
After overflow
3
arguments for system()
2
return address for system()
1
EIP
local variables
&"/bin/sh"
&exit()
&system()
16 A's
overflow direction
libc
0xb7def000
0xb7f52968
"/bin/sh"
0xb7e06c30
exit()
0xb7e13870
system()
0xb7dd6000

Race Condition vulnerability and attack

TCS 591 : Unit 2

What Is a Race Condition Vulnerability?

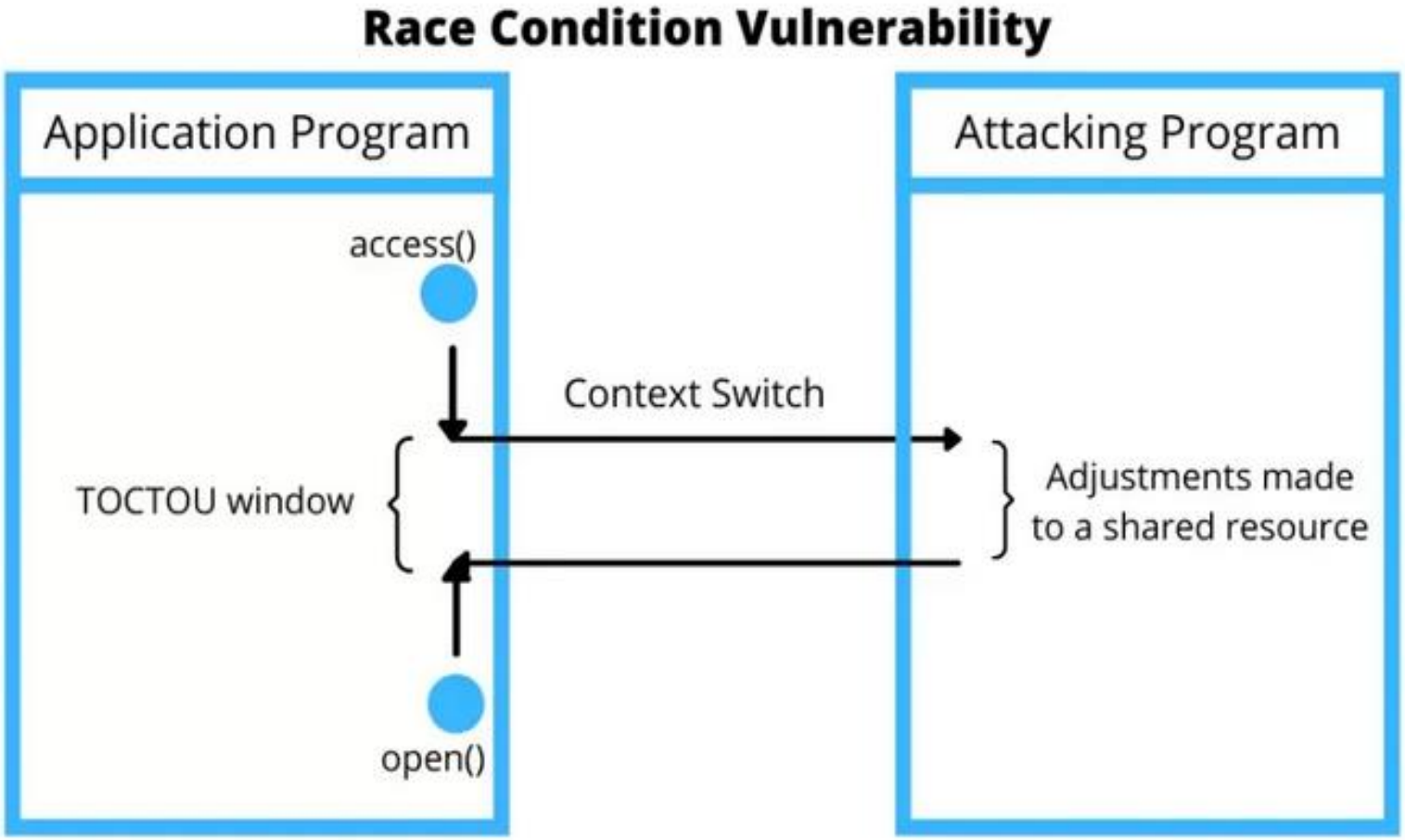
- A race condition attack happens when a computing system that's designed to handle tasks in a specific sequence is forced to perform two or more operations simultaneously.
- Other names used to refer to this vulnerability include Time of Check/Time of Use or **TOC/TOU** attacks.

What Is a Race Condition Vulnerability?

A race condition is when a system uses a resource that can be accessed concurrently (by the system itself or others). An attacker is able to take advantage of the **time gap** between:

1. when the system performs a security control on the shared resource;
2. when the system performs an operation on this shared resource.

If the attacker manages to change the shared resource between 1 and 2, then a race condition vulnerability is present


Race Condition Vulnerability
Application Program
Attacking Program
access()
Context Switch
TOCTOU window
Adjustments made
to a shared resource
open()

RACE CONDITION VULNERABILITY

Thread A	Thread B	Wallet Balance
Access the Application		
	Login to Application	
Read Available balance		100
	Withdraw Amount 10 Euros	100
Current Balance		90
Read Available Balance		90
	Withdraw Amount 20 Euros	90
Current Balance		90

Thread A	Thread B	Wallet Balance
Access the Application		
	Login to Application	
Read Available balance		100
	Withdraw Amount 10 Euros	100
Current Balance		90
Read Available Balance		90
	Withdraw Amount 20 Euros	90
Current Balance		70

NORMAL CONDITION

Race Condition : Real Life Example

ATMs are designed to handle one withdrawal at a time of your remaining balance.

Let's say your remaining balance is Rs 50 and you arrange with a friend to withdraw your Rs 50 remaining balance from **two different ATMs** at the same time.

If you and your friend are successful in withdrawing Rs 50 each simultaneously from two different ATMs then you have successfully exploited a race condition attack on your bank

Race Condition Attack

- Race conditions can be found in many common attack vectors, including web applications, file systems, and networking environments.
- The good news is that, because they rely on attackers exploiting a system's process during a very short window of time, race conditions are somewhat rare.
- Most attackers will instead focus on easier types of attack, such as SQL injection vulnerabilities.

Race Condition Attack

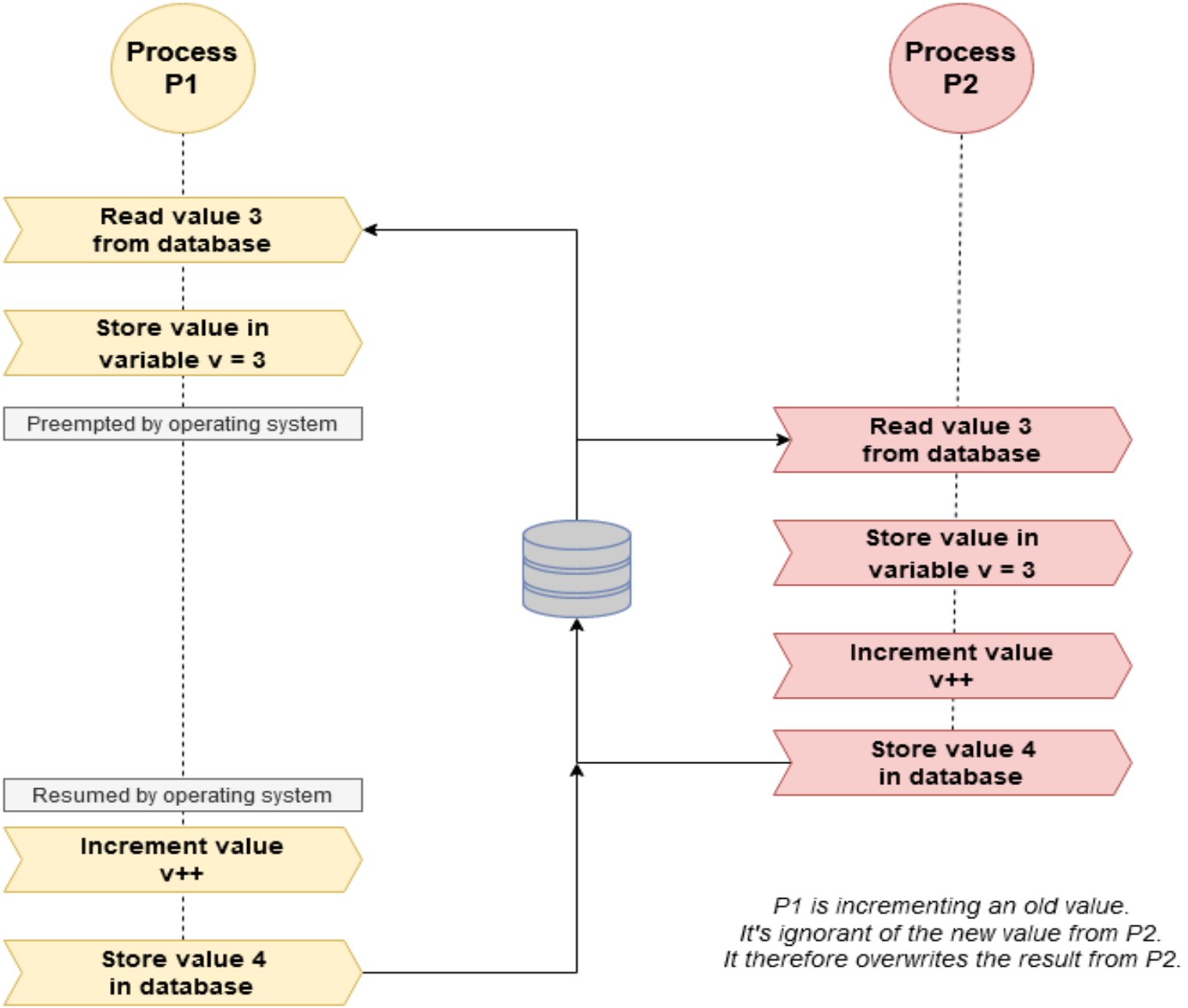
- Race condition attacks can lead to serious leaks and grant attackers complete access to secured systems, confidential data, and other information they shouldn't be able to view.
- Just a few of the types of systems that can be targeted with this kind of attack are access control list (ACL), a payroll or human resources database, a transactional system, a financial ledger, or any other data repository.

Race Condition Attack

- Almost all programs and web applications today use what is called **"multi-threaded"** processing, where they are capable of executing multiple actions at once.
- Although this allows applications to be significantly faster, it does introduce the potential for errors if more than one process (or "thread") tries to access the same data at the same time.

Race Condition Attack ：Example

- when a SQL system performs an update to a database, it creates a temporary file during the update process.
- It's this temporary file that eventually replaces the data in the database.
- With a well-timed attack, malicious users can replace the temporary file for a SQL update of an administrative access table with one of their own, granting themselves administrator privileges to the system.



Illustrating software race condition. Source: Devopedia 2020.

Preventing Race Conditions

- To prevent race conditions from occurring you must make sure that the critical section is executed as an atomic instruction.
- That means that once a single thread is executing it, no other threads can execute it until the first thread has left the critical section.
- Race conditions can be avoided by proper thread synchronization in critical sections. Thread synchronization can be achieved using a synchronized block of Java code.
- Thread synchronization can also be achieved using other synchronization constructs like locks or atomic variables like java.util.concurrent.atomic.AtomicInteger

Preventing Race Conditions

- The usual solution to avoid race condition is to **serialize access to the shared resource.**
- If one process gains access first, the resource is **"locked"** so that other processes have to wait for the resource to become available.
- Even if the operating system allows other processes to execute, they will get blocked on the resource.
- This allows the first process to access and update the resource safely. Once done, it will **"release"** the resource. One of the processes waiting on the resource will now get its chance to access the resource.
- Code protected this way using locks is called **critical section**. The idea of granting access to only one process is called **mutual exclusion**.

Dirty COW (Dirty copy-on-write)

TCS 591 : Unit 2

What is a Dirty COW ?

- Dirty COW (Dirty copy-on-write)
- Dirty COW is a computer security vulnerability that affects all Linux-based systems, including "Android" devices(that used older versions of the Linux kernel created before 2018)
- It is a local **privilege escalation** bug that exploits a **race condition** in the implementation of the copy-on-write mechanism in the kernel's memory-management subsystem
- Dirty Cow was one of the first security issues transparently fixed in **Ubuntu** by the Canonical Live Patch service

What is a Dirty COW ?

- Dirty COW vulnerability is a type of privilege escalation exploit, which essentially means that it can be used to gain **root-user access** on any Linux-based system
- Dirty COW was a vulnerability in the Linux kernel. It allowed processes to **write to read-only files**. This exploit made use of a **race condition** that lived inside the kernel functions which handle the copy-on-write (COW) feature of memory mappings

What happen in a Dirty COW attack

- In Dirty COW Malicious programs can potentially set up a **race condition** to turn a read-only mapping of a file into a writable mapping.
- Thus, an underprivileged user could utilize this flaw to elevate their privileges on the system.
- By gaining root privileges, malicious programs obtain unrestricted access to the system.
- From there on, it can modify system files, deploy keyloggers, access personal data stored on your device, etc.

What Systems Are Affected By Dirty COW vulnerability ?

- Dirty COW vulnerability affects all versions of the Linux Kernel since version 2.6.22, which was released in 2007.
- According to Wikipedia, the vulnerability has been patched in kernel versions 4.8.3, 4.7.9, 4.4.26 and newer.
- A patch was released in 2016 initially, but it didn't address the issue fully, so a subsequent patch was released in November 2017.
- To check your current kernel version number, you can use the following command on your Linux-based system
  - $ uname - r
  - Major Linux distros like Ubuntu, Debian, ArchLinux have all released the appropriate fixes.

How Dirty COW Affects Android Devices

- ZNIU is the **first malware** for Android based on the Dirty COW vulnerability. It can be utilized to root any Android devices up to Android 7.0 Nougat.
- While the vulnerability itself affects all versions of Android, ZNIU specifically affects Android devices with the ARM/X86 64-bit architecture.

Format String vulnerability and attack

TCS 591 Unit 2

What Are Format String Vulnerabilities ?

- Format strings are used in many programming languages to insert values into a text string. In some cases, this mechanism can be abused to perform buffer overflow attacks, extract information or execute arbitrary code
- The Format String exploit occurs when the submitted data of an input string is evaluated as a command by the application.
- In this way, the attacker could execute code, read the stack, or cause a segmentation fault in the running application, causing new behaviours that could compromise the security or the stability of the system

What Are Format String Vulnerabilities ?

- Format String attacks alter the flow of an application. They use string formatting library features to access other memory space.
- Vulnerabilities occurred when the user-supplied data is deployed directly as formatting string input for certain C/C++ functions (e.g., fprintf, printf, sprintf, setproctitle, syslog, ...).
- Format String attacks are related to other attacks in the Threat Classification: Buffer Overflows and Integer Overflows.
- All three are based on their ability to manipulate memory or its interpretation in a way that contributes to an attacker's goal.

What Are Format String Vulnerabilities ?

- To understand the attack, it's necessary to understand the components that constitute it.

The **Format Function** is an ANSI C conversion function, like **printf, fprintf**, which converts a primitive variable of the programming language into a human-readable string representation.

The **Format String Parameter**, like **%x %s** defines the type of conversion of the format function.

Format String

Format String

`printf()` - To print out a string according to a format.

```
int printf(const char *format, …);
```

The argument list of `printf()` consists of :

- One concrete argument format
- Zero or more optional arguments

Hence, compilers don't complain if less arguments are passed to `printf()` during invocation.

Example

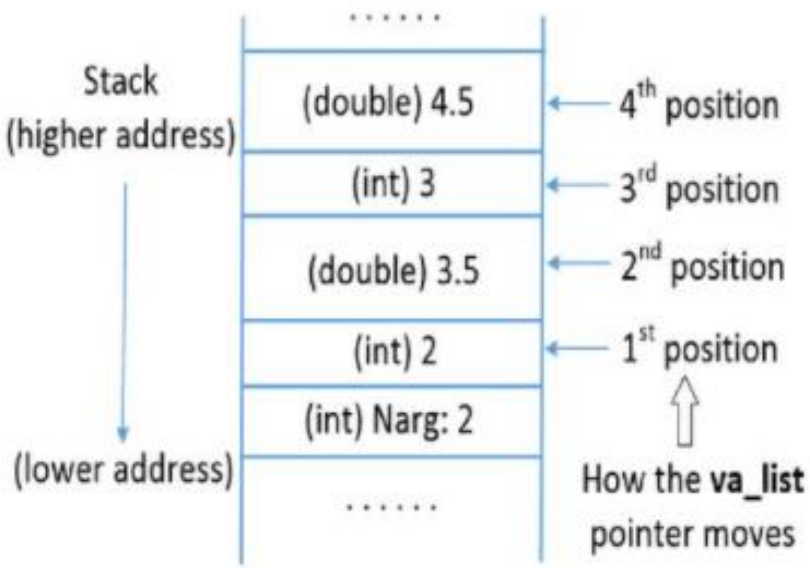
Access Optional Arguments

```
#include <stdio.h>
#include <stdarg.h>

int myprint(int Narg, ... )
{
  int i;
  va_list ap;                                          ①

  va_start(ap, Narg);                                  ②
  for(i=0; i<Narg; i++) {
    printf("%d  ", va_arg(ap, int));                   ③
    printf("%f\n", va_arg(ap, double));                ④
  }
  va_end(ap);                                          ⑤
}

int main() {
  myprint(1, 2, 3.5);                                  ⑥
  myprint(2, 2, 3.5, 3, 4.5);                          ⑦
  return 1;
}
```

- myprint() shows how printf() actually works.
- Consider myprintf() is invoked in line 7.
- va_list pointer (line 1) accesses the optional arguments.
- va_start() macro (line 2) calculates the initial position of va_list based on the second argument Narg (last argument before the optional arguments begin)

Example

Access Optional Arguments



- va_start() macro gets the start address of Narg, finds the size based on the data type and sets the value for va_list pointer.
- va_list pointer advances using va_arg() macro.
- va_arg(ap, int) : Moves the ap pointer (va_list) up by 4 bytes.
- When all the optional arguments are accessed, va_end() is called.

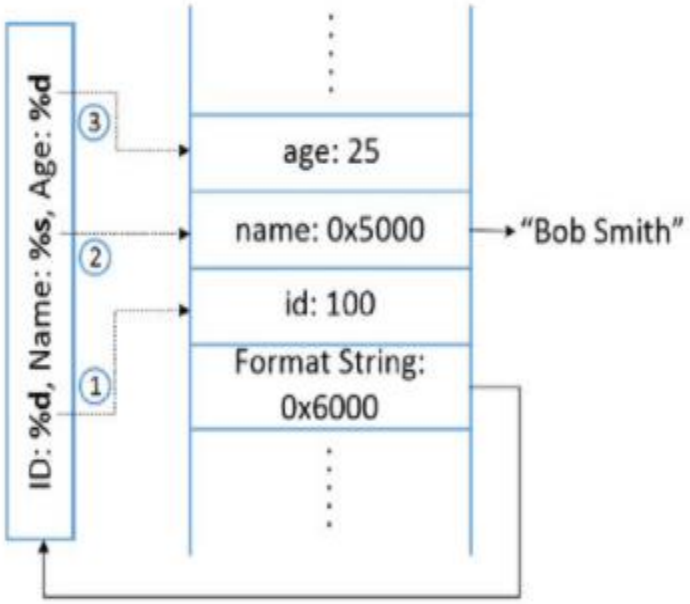
How `printf()` Access Optional Arguments

```
#include <stdio.h>

int main()
{
   int id=100, age=25; char *name = "Bob Smith";
   printf("ID: %d, Name: %s, Age: %d\n", id, name, age);
}
```

- Here, `printf()` has three optional arguments. Elements starting with "%" are called format specifiers.
- `printf()` scans the format string and prints out each character until "%" is encountered.
- `printf()` calls **va_arg()**, which returns the optional argument pointed by **va_list** and advances it to the next argument.

Example

How `printf()` Access Optional Arguments



- When printf() is invoked, the arguments are pushed onto the stack in reverse order.
- When it scans and prints the format string, printf() replaces %d with the value from the first optional argument and prints out the value.
- va_list is then moved to the position 2.

Format String Vulnerability

Format String Vulnerability

```
printf(user_input);
```

```
sprintf(format, "%s %s", user_input, ": %d");
printf(format, program_data);
```

```
sprintf(format, "%s %s", getenv("PWD"), ": %d");
printf(format, program_data);
```

In these three examples, user's input (user_input) becomes part of a format string.

What will happen if **user_input** contains format specifiers?

Vulnerable Code

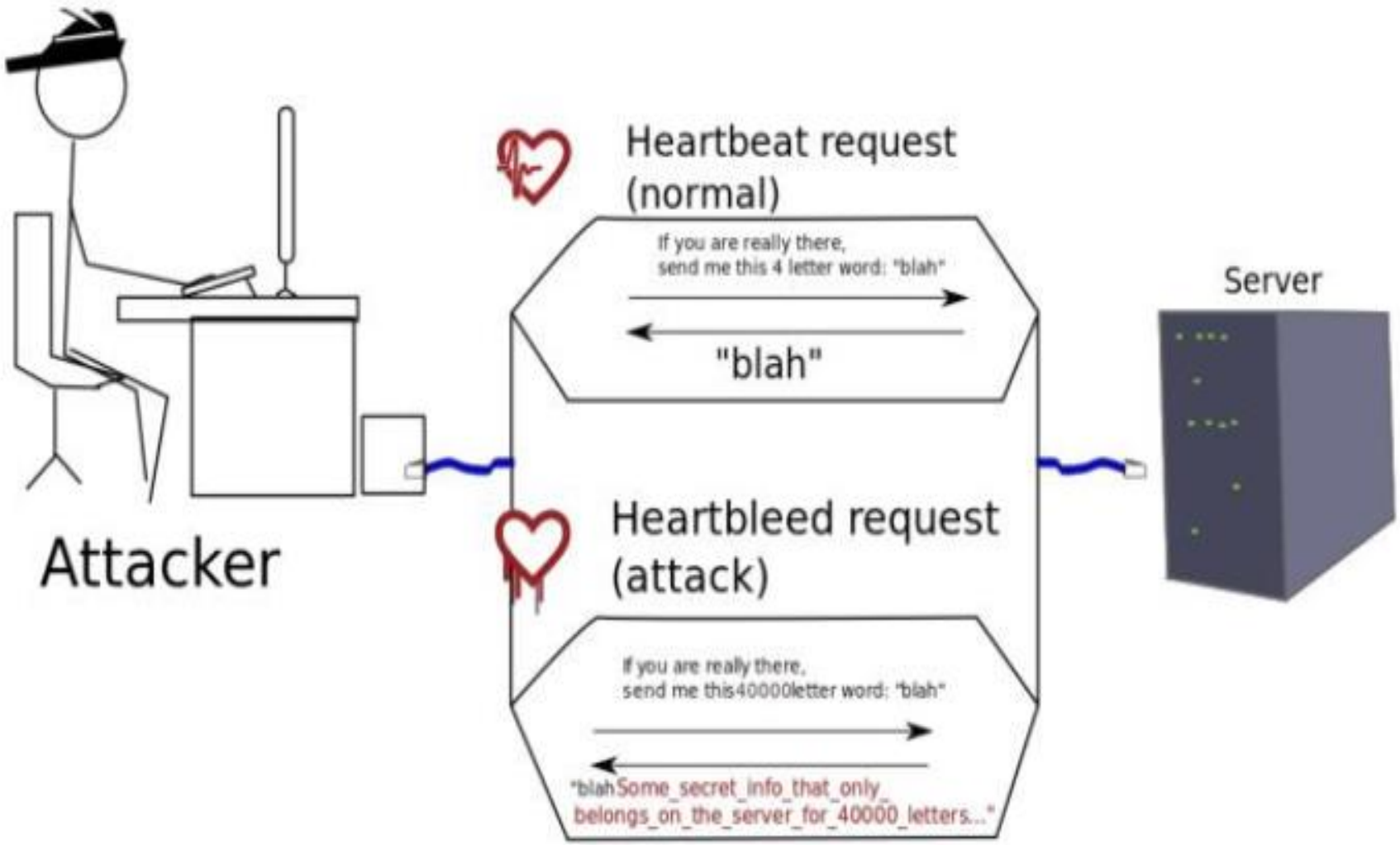
Vulnerable Code

```
#include <stdio.h>

void fmtstr()
{
    char input[100];
    int var = 0x11223344;

    /* print out information for experiment purpose */
    printf("Target address: %x\n", (unsigned) &var);
    printf("Data at target address: 0x%x\n", var);

    printf("Please enter a string: ");
    fgets(input, sizeof(input)-1, stdin);

    printf(input); // The vulnerable place    ①

    printf("Data at target address: 0x%x\n",var);
}

void main() { fmtstr(); }
```

Difference between Buffer Overflow and Format String exploits

- In buffer overflow, the programmer fails to keep the user input between bounds, and attackers exploit that to overflow their input to write to adjacent memory locations.
- But in format string exploits, user-supplied input is included in the format string argument. Attackers use this vulnerability and control the location where they perform arbitrary writes.

Heartbleed & Shellshock


Heartbeat request
(normal)
If you are really there,
send me this 4 letter word: "blah"
"blah"
Server
Attacker
Heartbleed request
(attack)
If you are really there,
send me this40000letter word: "blah"
"blahSome_secret_info_that_only_
belongs_on_the_server_for_40000_letters..."

Heartbleed

- The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library.
- This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet.
- SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).

Heartbleed

- The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the **OpenSSL** software. This compromises the **secret keys** used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content.
- When it is exploited it leads to the leak of **memory contents from the server** to the client and from the client to the server.
- CVE-2014-0160 is the official reference to this bug. CVE (Common Vulnerabilities and Exposures)

- Shellshock is effectively a Remote Command Execution vulnerability in BASH

- The vulnerability relies in the fact that BASH incorrectly executes trailing commands when it imports a function definition stored into an environment variable

Shellshock Vulnerability

- Shellshock is a computer bug that exploits the vulnerability in the UNIX command execution **shell-bash** to facilitate hackers to take control of the computer system remotely and execute arbitrary code, which affects UNIX based operating systems, including Linux and Mac OS.
- Most computers and 'IoT' enabled smart devices like routers, Wi-Fi radios, webcams and even smart bulbs running on Linux OS are most likely affected.

Shellshock Vulnerability

- Shellshock is effectively a Remote Command Execution vulnerability in BASH
- The vulnerability relies in the fact that BASH incorrectly executes trailing commands when it imports a function definition stored into an environment variable