Video Lecture # 40
Exploiting Buffer Overflow Vulnerability
Part-III

Course: SYSTEM PROGRAMMING

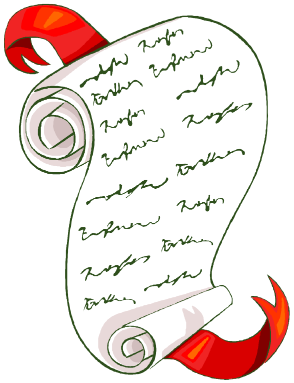
Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab

Source Code files available at: `https://bitbucket.org/arifpucit/spvl-repo/src`
Lecture Slides available at: `http://arifbutt.me`

Today's Agenda

- Finding vulnerabilities in executables
- Crafting input string to programs to shift the flow of control
- Writing Shell codes
- Injecting shellcode in the buffer
- Injecting shellcode in environment variables
- Executing shell codes inside gdb
- Executing shell codes outside gdb



Exploiting a Buffer Overflow Vulnerability

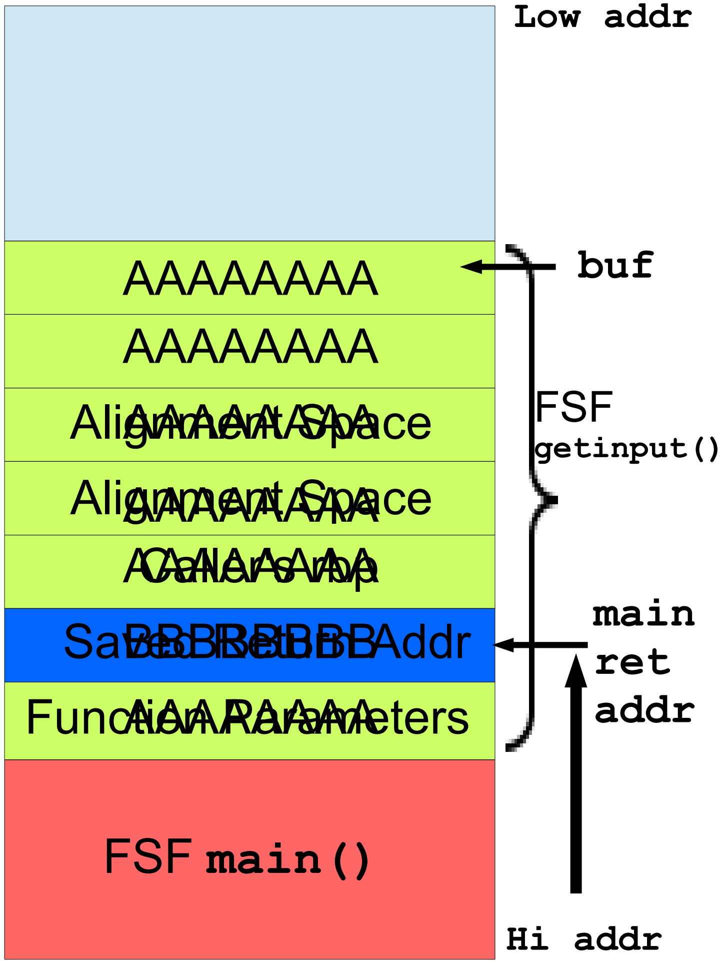
PART-I

- Find and understand the vulnerability in the program
- Give it an input string such that the control of flow move to some piece of code of our choice within the code section

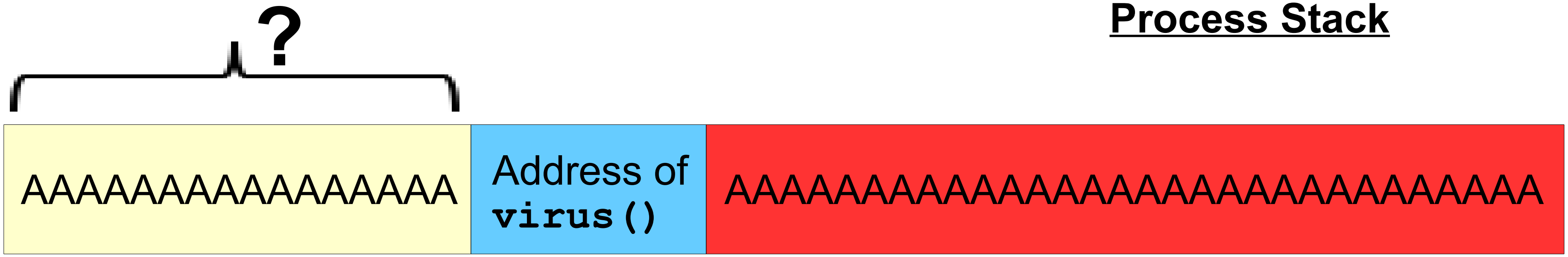
Change the flow of execution to `virus()`

```
int getinput(){
   char buf[10];
   int rv = read(0, buf, 1000);
   printf("\nNumber of bytes read are %d",rv);
   return 0;
}

int main(int argc, char *argv[]) {
   getinput();
   return 0;
}
int virus(){
   printf("Let us Hack Planet Earth with Arif
            Butt.\n");
   exit(0);
}
```



Process Stack



Changing Control of Flow
Proof of Concept

Exploiting a Buffer Overflow Vulnerability

PART-I

- Find and understand the vulnerability in the program
- Give it an input string such that the control of flow move to some piece of code of our choice within the code section

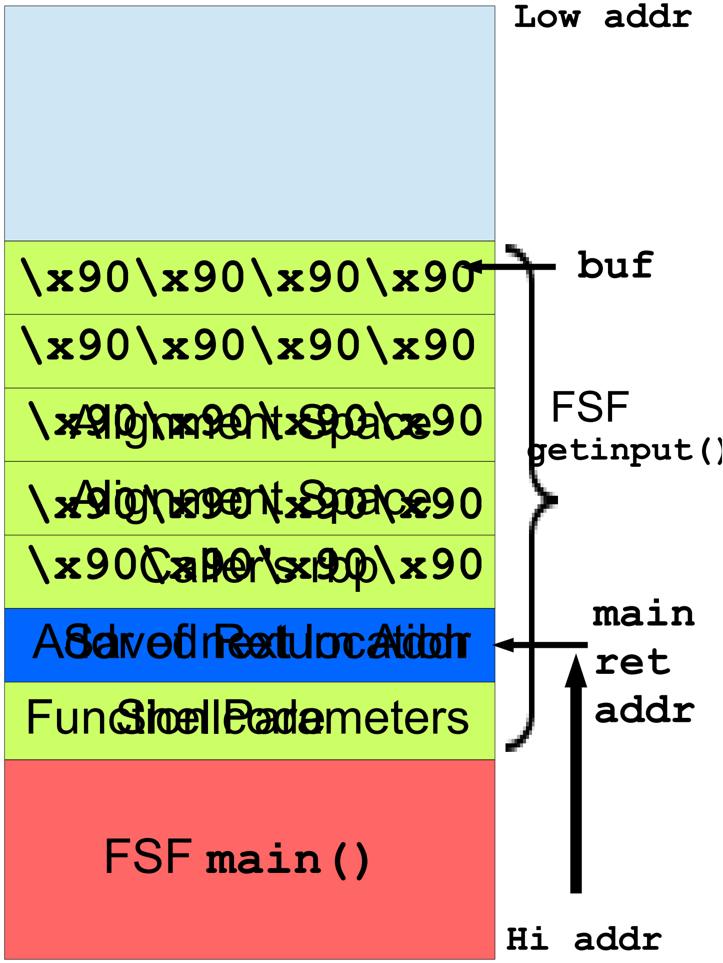
PART-II

- Write / Get the *shellcode*
- Craft the input string such that the control of flow shifts to your *shellcode* residing on the stack
- Inject the *shellcode* by giving this input string to the vulnerable program
- Test it inside the debugger
- Test it outside the debugger

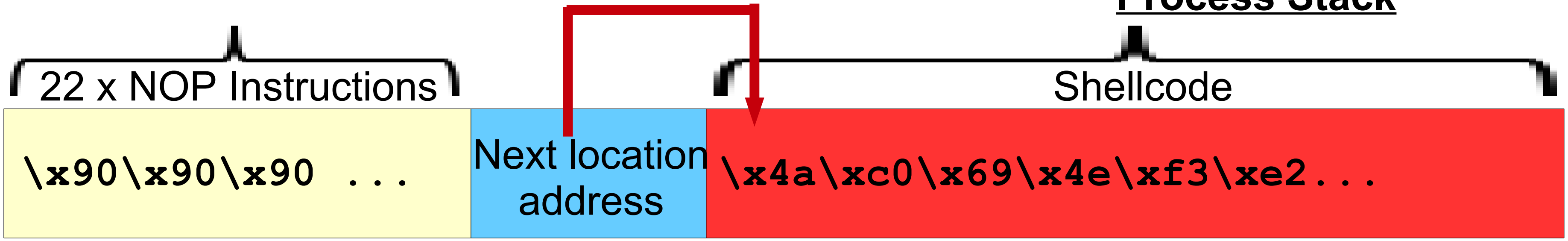
Change the flow of execution to Shellcode

```
int getinput(){
   char buf[10];
   int rv = read(0, buf, 100);
   printf("\nNumber of bytes read are %d",rv);
      return 0;
}

int main(int argc, char *argv[]) {
   getinput();
   return 0;
}
int virus(){
   printf("Let us Hack Planet Earth with Arif
               Butt.\n");
   exit(0);
}
```



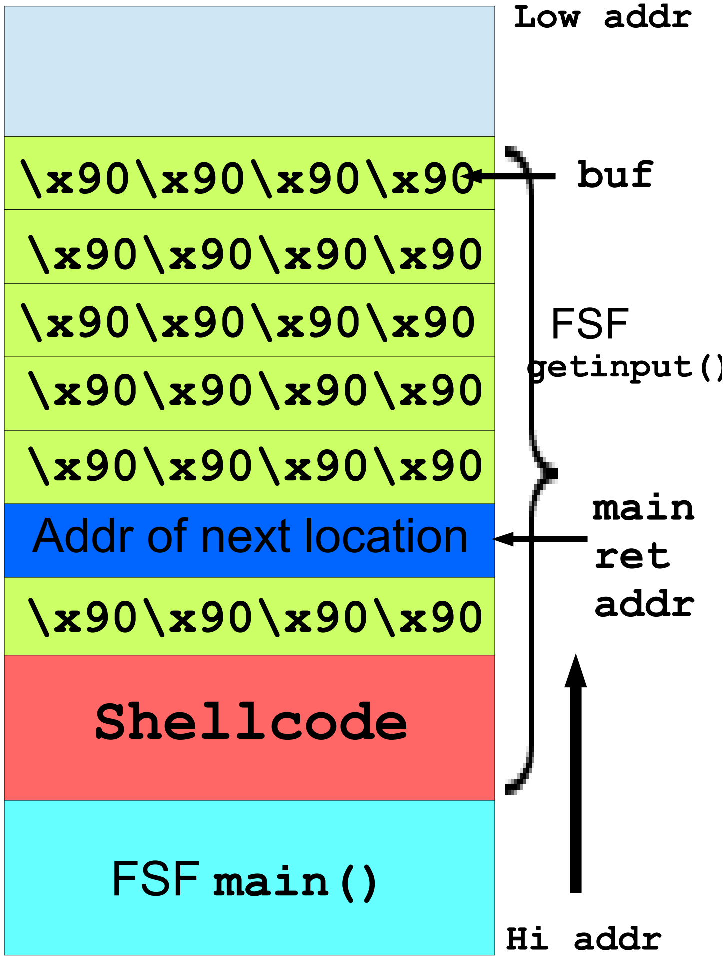
Process Stack



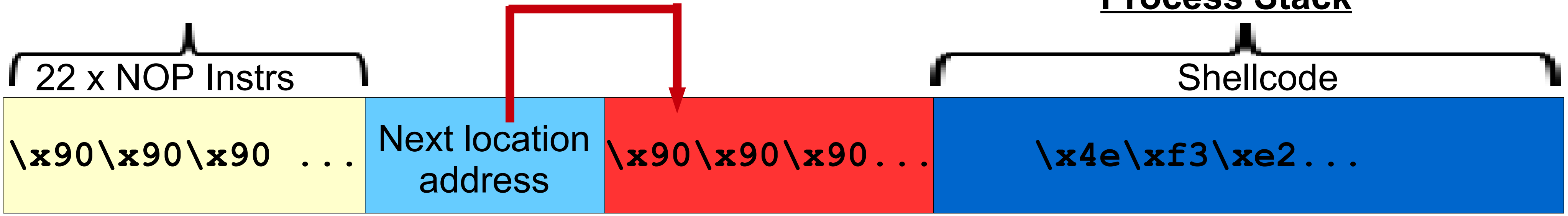
Change the flow of execution to Shellcode

```
int getinput(){
   char buf[10];
   int rv = read(0, buf, 100);
   printf("\nNumber of bytes read are %d",rv);
      return 0;
}

int main(int argc, char *argv[]) {
   getinput();
   return 0;
}
int virus(){
   printf("Let us Hack Planet Earth with Arif
               Butt.\n");
   exit(0);
}
```



Process Stack



Injecting and Running Shellcode Proof of Concept

HOME TASKS

Things to Do

- Use the executable of vulnerable echo server program uploaded on the bitbucket account. Carry out the vulnerability analysis, craft the input string and inject the shell code that will perform following tasks as discussed in the Video Session # 39:
  - TCP bind shell
  - TCP reverse shell

Code Integrity Protection

- The **code integrity** property is ensured by making the process stack, heap, and data sections as non-executable and the text section as non-writable, which is the default settings in most modern operating systems
- To make the stack executable we can set the NX (Non-Executable) bit in x86. A programmer can set this bit during linking phase by giving the `-z noexecstack` option to `gcc`
- The text section is also by default non-writable, and this is also enforced by memory access permissions using virtual memory (in the page table)

Bypassing Code Integrity Protection

- You can **bypass Code Integrity Protection** by using a class of attacks known as `return-to-libc`, in which you divert the control flow to C library function `system()` with the `/bin/sh` parameter. The fact that the vast majority of programs are linked with the C library makes this pretty easy. This technique of attacks is also known as Return-Oriented-Programming (ROP), in which you reuse existing code in the program, the attacker may reuse small pieces of program code called "gadgets" to execute arbitrary (turing-complete) operations
- You can **bypass Code Integrity Protection** by making the stack executable at run-time, e.g., by calling `mprotect()`, which is used to change the access protections for the calling process's memory page containing the stack

Address Space Layout Randomization

- ASLR randomizes the address of stack, heap and hared library sections in a process address space. Every time a program executes it is given different addresses, thus preventing an attacker from reliably jumping to an exploited function in memory
- Linux allows 3 options for its ASLR implementation that can be configured in `/proc/sys/kernel/randomize_va_space` file. Writing 0, 1, or 2 to this will results in the following behaviors:
  - → **0**: deactivated
  - → **1**: random stack, vdso, libraries; data is after code section
  - → **2**: random data too
- → The **stack** is easily randomizable, as all stack addresses are relative to `rsp` or `rbp`. Similarly **data** section can also be randomized, if address of data segment is set to a random value
- The **Code** can only be randomized by compiling the program as Position Independent Code/Position Independent Executable. This is the default for shared libraries, but otherwise executable code is usually placed at fixed addresses. Note that randomization occurs at **load-time**, which means that the segment addresses **do not** change while the process is running

Bypassing - Address Space Layout Randomization

- **Bruteforce**. If the attacker is able to inject payloads multiple times without crashing the application, they can bruteforce the address they are interested in (e.g., a target in libc). Otherwise, they can just run the exploit multiple times until they guess the correct target
- **NOP sled**. In the case of shellcodes, a longer NOP sled will maximize the chances of jumping inside it and eventually reaching the exploit code even if the stack address is randomized
- **Restrict entropy**. There are various ways of reducing the entropy of the randomized address. For example, the attacker can decrease the initial stack size by setting a huge amount of dummy environment variables
- **Information leak**. The most effective way of bypassing ASLR is by using an information leak vulnerability that exposes a randomized address, or at least parts of it. The attacker can also dump parts of libraries (e.g., libc) if they are able to create an exploit that reads them. This is useful in remote attacks to infer the version of the library, downloading it from the web, and thus knowing the right GOT offsets for other functions (not originally linked with the binary)

Stack Protection: Canaries

- A stack canary is a known value or word that is placed just below the ruturn address on the stack to monitor buffer overflow. A copy of this word is saved some where else as well. When the hacker overwrites the return address using a buffer overflow the canary will also be overwritten. When a function calls return this canary value is compared with a a saved else where copy and a mismatch indicates that the stack is over wirtten. To disable this security option we can compile the program using `-fno-stack-protector` option to `gcc`
- Stack canaries only protect against buffer overflows. Arbitrary memory writes (e.g. to offsets that can be controlled by the attacker) may be crafted so that they do not touch the canary value. Guessing the canary value, e.g. through an information leak or through brute force, is possible and will bypass the attack. **Search out more options that can be used to override this protection mechanism**

HAPPY LEARNING