HO# 3.7 Mitigation Techniques for BoF Vulnerability

Overview

Buffer Overflow (BoF) vulnerabilities occur when a program writes more data to a buffer than it can hold, leading to potential overwriting of adjacent memory. This can result in crashes, data corruption, or security exploits, including arbitrary code execution. Various security measures and mitigation techniques have been developed to prevent or mitigate the risk of buffer overflows. We have categorized them into *developer-based*, *OS-based*, and *compiler-based* techniques. By employing a combination of these techniques, systems can be hardened against buffer overflow attacks and other memory corruption vulnerabilities.

Developer-Based Techniques

These are techniques that developers can/should use during the design and coding phases to reduce the chances of a buffer overflow:

1. **Input Validation:** Always validate input data to ensure it conforms to expected lengths, types, and formats. For example:
   - A developer should check data lengths before copying them to buffers inside the memory. (e.g., `strlen` to ensure the length is within bounds).
   - It is always a good practice to set the size of buffer yourself, and do not let users set the length.
   - Validate user input (e.g., checking for null-termination or controlling the size of arrays).

2. **Use Safe C Functions:** Use safer versions of functions that deal with buffers, which limit the amount of data written to a buffer. It is best practice to replace unsafe C functions like `strcpy`, `strcat`, `sprintf`, `scanf`, and `gets` with their safer counterparts such as `strncpy`, `strncat`, `snprintf`, `fscanf`, and `fgets` respectively, that allow you to specify buffer sizes. Similarly, in C++, use `std::vector` or `std::string` instead of raw arrays for dynamic memory management, which handles resizing and bounds checking. Moreover, while using C's dynamic memory allocation (e.g., `malloc`, `calloc`, `realloc`, and `free`) functions, ensure the allocated buffer size is correct, avoid fixed size buffers, and initialize pointers and check for null pointers before dereferencing.

3. **Use of Safe Libraries:** Instead of linking your C program with standard C library, you can use `libsafe.so`, which is a shared library that intercepts and replaces dangerous C functions known to cause buffer overflows. By linking your application with `libsafe`, it hooks into the program and replaces unsafe functions with safer alternatives. Although it was a useful tool, today's compilers and libraries offer more integrated and modern security mechanisms.

4. **Use of Safer Programming Languages:** Use safer programming languages if you have choice. For example, *Rust* is a modern systems programming language that enforces memory safety without a garbage collector. Rust's ownership and borrowing system guarantees that common vulnerabilities (e.g., buffer overflows, use-after-free) do not occur.

OS-Based Techniques

No-eXecute (NX) / Data Execution Prevention (DEP)

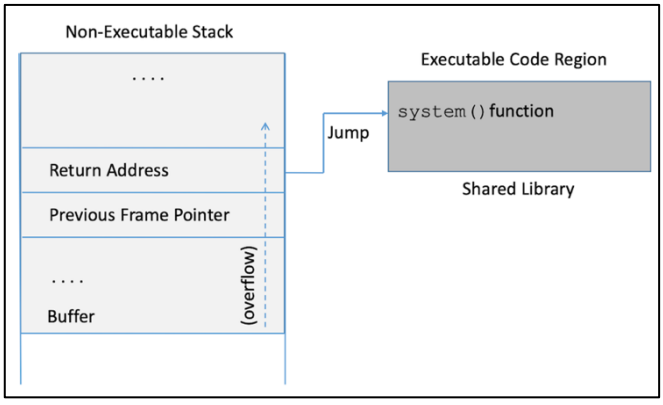
Dear students buffer overflow attacks in general and code injection attacks, in particular, are possible due to the common memory space for both code and data (Von Neumann architecture). The buffer overflow attack we have discussed so far depends on the execution of the shellcode, which is placed on the stack. The stack is mostly used for storing data, so **a $100 question is: Do we need to make the data memory (stack or heap) executable? and the answer is NO**. For most of the regular applications, there is no need to give execute permissions to data memory, which will prevent code injection attacks and can make the BOF attacks difficult.

The `No-eXecute (NX)` bit is a technology used in the hardware of CPUs to separate code from data. The Advanced Micro Devices (AMD) has introduced NX bit in `x64` processors to reduce the BOF attacks. While using NX bit, OS marks certain regions (stack and heap) of memory as non-executable for security purposes. It is supported by most modern operating systems, including Windows (DEP), Linux (NX bit), and macOS. While compiling a C program you can set/reset it using `-z` flag of `gcc`. OR on Linux you can use the `$ execstack -s ./a.out` or `$ execstack -c ./a.out` command to execute the binary with the bit set/clear respectively in the header of ELF binary.

```
//3.6/myshell.c
#include <stdio.h>
#include <string.h>
int main(){
    char code[] = "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c"
                  "\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52"
                  "\x57\x54\x5e\xb0\x3b\x0f\x05";

    printf("Length:%d bytes\n", strlen(code));

    int(*foo)() = (int(*)())code;

    foo();

    return 0;
}
```

```
// Make the stack executable
$ gcc -z execstack myshell.c -o myshell
$ ./myshell
Length: 27 bytes
$                    <-- Got a new shell


// Make the stack non-executable
$ gcc -z noexecstack myshell.c -o myshell
$ ./myshell
Length: 27 bytes
Segmentation fault (core dumped)
```

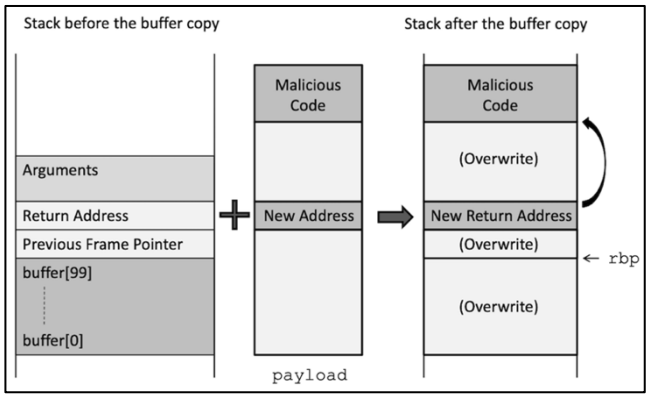
How secure is this counter measure?



Well, this will certainly defeat the technique using which we have exploited the BOF vulnerability by placing the shellcode inside the stack. However, we can switch control to *Program's own code*, *Library code*, or *OS code*. Jumping to an application's code is not very useful, and jumping to kernel code is not possible as it is protected. So, we are left with only one option and that is jumping to library code. The `return-into-libc` attack is a category of attack, where instead of jumping to our injected shellcode on the stack, we jump to some existing library code. So, we can find some function (e.g., `system`, `execve`) inside the `glibc` library, which allows us to run a shell program. The main task to do in such type of attack is to find the address of `system()` function, the address of the `/bin/sh` program or the string itself inside the memory and how to pass this string to the `system` function. `RILC` attack is a subset of Return-Oriented-Programming (`ROP`) attacks and focuses on calling functions already present in `libc`, while the `ROP` attacks actually chain smaller code gadgets. More on this later ☺

Address Space Layout Randomization (ASLR)

Dear students, we all know that in order to launch a BOF attack, the attacker needs to know the stack address where the return address is saved, as well as the stack address where the malicious code is to be injected. If the location where the `rbp` register is pointing to in a Function Stack Frame is known to us or if the starting address of buffer is known to us, we can guess the location of the saved return address and thus can craft an input string that will successfully launch the BOF attack. **A $100 question is: Do we need to have a fixed location for the start of the function stack frame? and the answer is NO**.



Address Space Layout Randomization (ASLR) is an operating system based defensive mechanism that protects against memory corruption vulnerabilities using a randomization strategy. ASLR is typically implemented at the operating system level and is supported by all modern operating systems like Linux, Windows, and macOS. Since, it randomizes the base addresses of stack and heap, so it makes the standard BOF attack that we have done quite difficult to perform. Moreover, since it also randomizes the addresses of shared libraries loaded in memory, so it also makes the `return-into-libc` attack difficult as well.

On Linux, one can set the ASLR setting using any of the following two commands by giving it a value of 0 (no randomization, 1 (partial randomization) or 2 (full randomization):

```
$ sudo sysctl kernel.randomize_va_space=0
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

To practically understand this, let us run the following C program multiple times with different settings of ASLR. The output is also shown for your understanding:

```
//3.6/aslr.c
#include <stdio.h>
#include <stdlib.h>
void main(){
   char x[12];
   char* y = malloc(sizeof(char)*12);
   printf("Address of buffer x (on stack): 0x%z\n", x);
   printf("Address of buffer y (on heap): 0x%z\n", y);
}
```

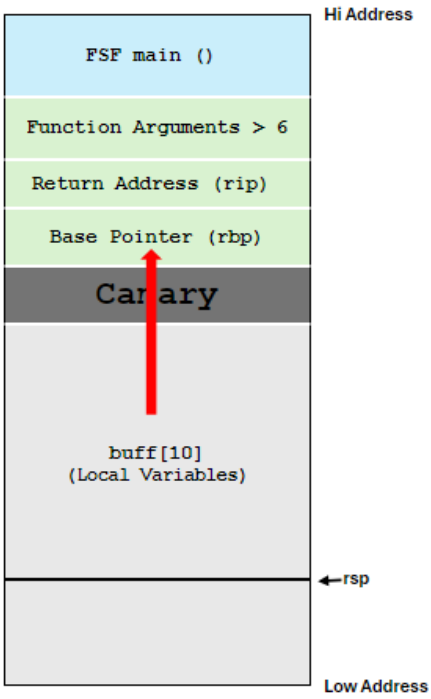
```
// Turn off randomization completely by setting the value to 0
$ sudo sysctl -w kernel.randomize va space=0
$ ./aslr
Address of buffer x (on stack): 0xffffddec
Address of buffer y (on heap) : 0x555592a0
$ ./aslr
Address of buffer x (on stack): 0xffffddec
Address of buffer y (on heap) : 0x555592a0

// Randomizing stack address only by setting the value to 1
$ sudo sysctl -w kernel.randomize va space=1
$ ./aslr
Address of buffer x (on stack): 0x44230bdc
Address of buffer y (on heap) : 0x8e7512a0
$ ./aslr
Address of buffer x (on stack): 0xcb84ae5c <-- Changed
Address of buffer y (on heap) : 0x8e7512a0

// Randomizing stack and heap address by setting the value to 2
$ sudo sysctl -w kernel.randomize va space=2
$ ./aslr
Address of buffer x (on stack): 0x32d0b12c
Address of buffer y (on heap) : 0xfeeb92a0
$ ./aslr
Address of buffer x (on stack): 0xd1ddc01c <-- Changed
Address of buffer y (on heap) : 0x8a4cd2a0 <-- Changed
```

How secure is this counter measure?

Even with ASLR in place, an attacker could still perform a *Return-Oriented Programming* (ROP) attack, which allows an attacker to execute code by chaining together short snippets of existing code (called "gadgets") found in the address space of the program or libraries. These gadgets are usually small instruction sequences ending with a `ret` instruction. The attacker then crafts a payload that overwrites the return addresses on the stack, redirecting execution to the chained gadgets. More on this later ☺

Compiler-Based Techniques

Stack Canary



Stack canary is a GCC compiler extension that is used to protect the application from BOF attacks by exploiting a function's return address. It was first introduced in Intel x86 as a patch to GCC compiler version 4.1 (2006). When a program calls a function, the return address is pushed in the FSF, and the stack canary/guard (a random value) is also placed on the stack right after the return address. This canary value is also saved somewhere else in a safer place. When the function returns after execution, the canary value is compared to the actual value. If both values are the same, it means the return address is protected. If a buffer overflow overwrites the canary value, the program detects the modification and terminates before the attack can proceed.

The `gcc` compiler from version 4.9 (2014) onwards provides following levels of stack protection using following flags:

- **`-fstack-protector`**: Enables basic stack protection, placing a canary value on the stack to detect overflows in functions with *local buffers* (especially those that might be vulnerable to overflow).
- **`-fstack-protector-all`**: Adds protection to all functions, not just those with local buffers.
- **`-fstack-protector-strong`**: A middle ground between `-fstack-protector` and `-fstack-protector-all`. It enables stack protection for functions that are more likely to be vulnerable (i.e., functions with local buffers or arrays), but not universally for all functions.
- **`-fstack-protector-explicit`**: Only the functions that are explicitly marked with the `__attribute__((stack_protector))` attribute in the code will have stack protection enabled.
- **`-fno-stack-protector`**: Disables stack protection (useful for situations where stack protection causes issues or is unnecessary).

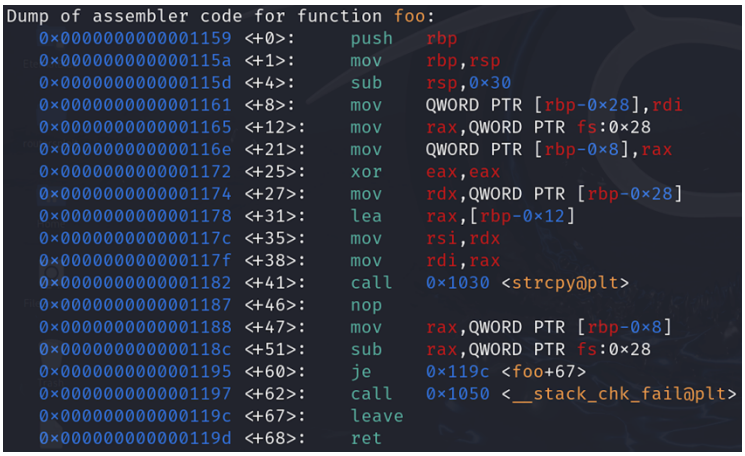
Let us run a program and have a proof of this concept:

```
//3.6/canary.c
#include <stdio.h>
#include <string.h>
void foo(char* str){
   char buff[10];
   strcpy(buf, str);
}
void main(int argc, char* argv[]){
   foo(argv[1];
   printf("Returned properly from foo\n");
   return 0;
}
```

```
// Without Stack Canary (large input)
$ gcc canary.c
$ ./a.out "this is a very long string"
Segmentation fault
$

// With Stack Canary (small input)
$ gcc -fstack-protector-all canary.c
$ ./a.out "hello"
Returned properly from foo

// With Stack Canary (large input)
$ ./a.out "this is a very long string"
***stack smashing detected ***: terminated
Aborted
$
```

The above output shows that on my `gcc` the stack guard/canary is by default disabled. So, when I compile above program using `-fstack-protector-all` flag, and have given it a large input, the program crashes, however, it will not overwrite the return address and the program will not return to the malicious code injected by the user.

Following screenshots display the disassembly of the function `foo`, after compiling with and without the stack guard. Compare the assembly to understand to have a clear understanding of working of stack guard ☺



How secure is this counter measure?
An attacker might use **information leaks** to determine the canary value (e.g., by exploiting format string vulnerabilities, buffer overflows, or side-channel attacks). Once the canary is known, the attacker can overwrite it along with the return address and proceed with exploiting the overflow. More on this later ☺

The Position Independent Code/Executable (PIC/PIE)

Dear students, we have already discussed that full ASLR randomizes the addresses of code section, data section, stack, heap, as well as the shared libraries. The Position Independent Code (**PIC**) and Position Independent Executable (**PIE**) are techniques where the `.text` section is made position-independent.

- The `-fPIC` flag is a *compilation* option of `gcc` that generates position-independent code (object files) for later creation of a shared library using the `--shared` flag of `gcc`. This allows the library to be loaded at any address in memory to be used by different programs.
- The `-fPIE` flag is a *compilation* option of `gcc` that generates position-independent code (object files) for later creation of a position-independent executable using the `-pie` flag of `gcc`. Remember, the `-pie` flag is a *linking* option of `gcc` that primarily affects the text segment of the executable by making it position-independent, allowing it to be loaded at any address in memory. Remember, the `-pie` flag does not affect the stack or heap directly.

In Linux implementation of ASLR, if the binary executable itself is not PIE compiled, using the `-no-pie` flag of `gcc`, it will limit the effectiveness of ASLR. The executable's text segment (code) may not be randomized. The proof of this concept is shown in the following screenshot. In the output, note that ASLR is randomizing the locations of stack and heap, however, when the executable is generated with `-no-pie` option, the program code is loaded at the same address each time. ☺

```
//3.6/pie.c
#include <stdio.h>
#include <stdlib.h>
void main(){
   char x[12];
   char* y = malloc(sizeof(char)*12);
   printf("Address of buffer x (on stack): 0x%z\n", x);
   printf("Address of buffer y (on heap): 0x%z\n", y);
   printf("Address of main (on .text): 0x%z\n", main);

}
```

```
// Full randomization and PIE enabled
$ sudo sysctl -w kernel.randomize va space=2
$ gcc -pie pie.c
$ ./a.out
Address of buffer x (on stack): 0xe4b2143c
Address of buffer y (on heap) : 0x223f32a0
Address of main     (on .text): 0x20f36149
$ ./a.out
Address of buffer x (on stack): 0xf99ca95c
Address of buffer y (on heap) : 0x193b12a0
Address of main     (on .text): 0x1873e149

// Full randomization and PIE disabled
$ sudo sysctl -w kernel.randomize va space=2
$ gcc -no-pie pie.c
$ ./a.out
Address of buffer x (on stack): 0xbc6135bc
Address of buffer y (on heap) : 0x14f12a0
Address of main     (on .text): 0x401136
$ ./a.out
Address of buffer x (on stack): 0x33fdcdbc
Address of buffer y (on heap) : 0x7b22a0
Address of main     (on .text): 0x401136 <---Not Changed
```

The Control Flow Protection (CFP)

Control Flow Protection works by introducing security mechanisms that prevent an attacker from taking control of the program's execution flow by manipulating function pointers, return addresses, or other control data. The protection ensures that the control flow of the program is checked at runtime, and any invalid or unexpected jumps (such as those caused by an attacker manipulating return addresses or function pointers) are blocked. The `-fcf-protection` flag of `gcc` provides a safeguard against following types of attacks:

- Buffer overflow attacks leading to control flow hijacking.
- Function pointer manipulation, where an attacker can manipulate a function pointer to point to malicious code.
- Return-Oriented Programming (ROP), where attacker manipulates the return address to redirect the program's execution to "gadgets" (small code snippets)
- Jump-Oriented Programming (JOP), which is similar to ROP, but uses jumps instead of returns.

The `gcc` compiler provides different levels of `-fcf-protection` with following levels:

- **`-fcf-protection=none`**: No control flow protection is applied, which is the default.
- **`-fcf-protection=ret`**: This applies protection only to `ret` instructions, which prevents attackers from manipulating return addresses, which is a common technique in ROP attacks. It ensures that the return address at the end of a function points to a valid location and not to an attacker-controlled location.
- **`-fcf-protection=full`**: This is the most robust protection level, that applies control flow protection to all indirect branches, including function calls, returns, and other jumps (like those using function pointers).

FORTIFY_SOURCE

The `-D_FORTIFY_SOURCE` macro in GCC is a feature designed to provide extra protection against certain types of common vulnerabilities, such as buffer overflows. While compiling your program, one can use `-D_FORTIFY_SOURCE=level` macro that enables compile-time or runtime BOF detection depending on the protection level (for functions like `strcpy`, `sprintf`, `memcpy` etc.) The `_FORTIFY_SOURCE` macro can be defined with different levels, each providing different amounts of protection:

- `-D_FORTIFY_SOURCE=0`: This is the default setting, in which there are no extra checks or optimizations to catch overflows or misuse of memory-related functions.
- `-D_FORTIFY_SOURCE=1`: This level performs some basic compile-time checks on certain functions to ensure they aren't misused.
- `-D_FORTIFY_SOURCE=2`: This is the most aggressive level of fortification, and increases the likelihood that dangerous operations (such as accessing memory beyond the bounds of a buffer) will trigger a compile-time or runtime error.

Disclaimer

The series of handouts distributed with this course are only for educational purposes. Any actions and or activities related to the material contained within this handout is solely your responsibility. The misuse of the information in this handout can result in criminal charges brought against the persons in question. The authors will not be held responsible in the event any criminal charges be brought against any individuals misusing the information in this handout to break the law.