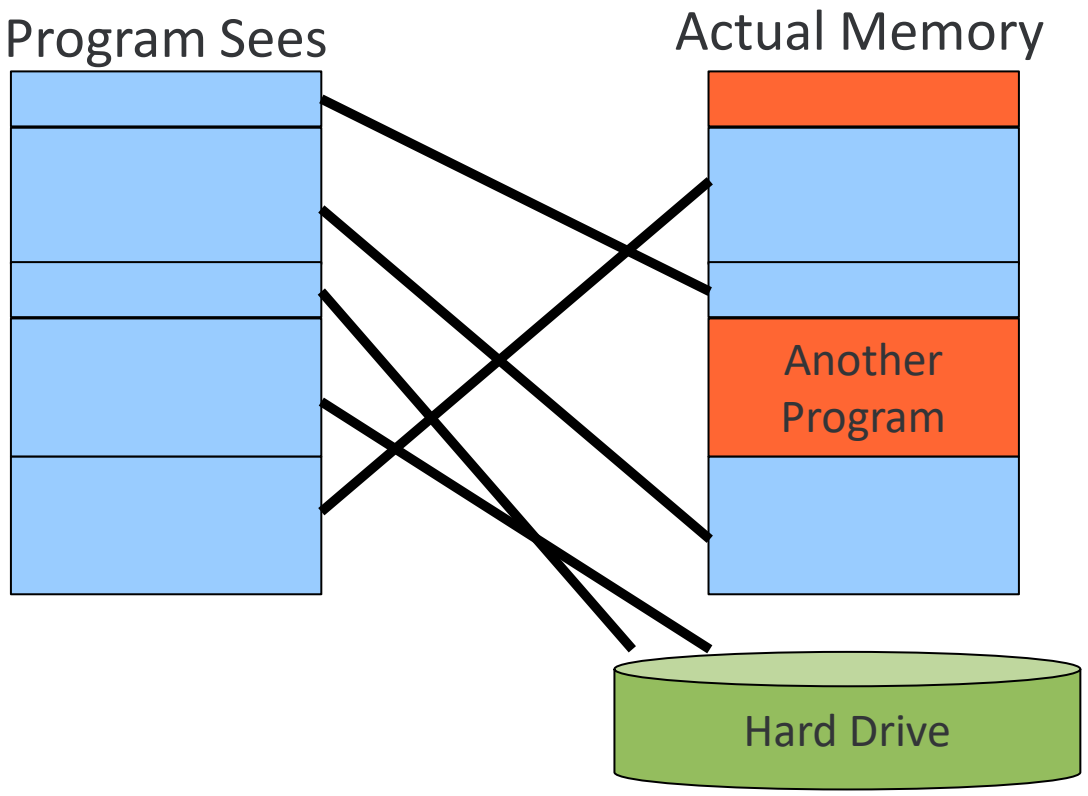
Buffer Overflow Attacks



What is an Exploit?

- An exploit is any input (i.e., a piece of software, an argument string, or sequence of commands) that takes advantage of a bug, glitch or vulnerability in order to cause an attack.
- An attack is an unintended or unanticipated behavior that occurs on computer software, hardware, or something electronic and that brings an advantage to the attacker.

What is an Exploit?

- An exploit is not necessarily a program.
- While it can be a program that communicates bad input to a vulnerable piece of software, it can also be just the bad input itself.
- Any bad input (or even valid input that the developer just failed to anticipate) can cause the vulnerable application to behave improperly.

Buffer Overflow Attack

- One of the most common OS bugs is a buffer overflow
  - The developer fails to include code that checks whether an input string fits into its buffer array.
  - An input to the running process exceeds the length of the buffer.
  - The input string overwrites a portion of the memory of the process.
  - Causes the application to behave improperly and unexpectedly.

Buffer Overflow Attack

- Since the stack grows downward, if you write past the end of the buffer, you can corrupt the content of the rest of the stack, if enough information is known about the program.
- Because of the nature of the address space, locally declared buffers are allocated on the stack and one could write over known register information and the return address.

Buffer Overflow Attack

- Effect of a buffer overflow
  - The process can operate on malicious data or execute malicious code passed by the attacker.
  - If the process is executed as root, the malicious code will be executing with root privileges.

Buffer Overflow – Simple Example

```
#include <string.h>

void foo(char *bar)
{
    char  c[12];
    strcpy(c, bar);  // no bounds checking
}

int main(int argc, char **argv)
{
    foo(argv[1]);
    return 0;
}
// a.out arg1tooooloooooongg
```

Buffer Overflow – Simple Example

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

void copyData(char* input) {
    char buffer[8];      // Fixed-size buffer with a capacity of 8 bytes
    strcpy(buffer, input);     // Copy user input to the buffer
    printf("Data copied: %s\n", buffer);
}

int main() {
    char userInput[20];
    printf("Enter data: ");
    scanf("%s", userInput);
    copyData(userInput);
    return 0;
}
```

Potential Vulnerabilities in copyData??

Buffer Overflow – Simple Example

```
void copyData(char* input) {
    char buffer[8];      // Fixed-size buffer with a capacity of 8 bytes
    strcpy(buffer, input);     // Copy user input to the buffer
    printf("Data copied: %s\n", buffer);
}
```

In this example, the *copyData* function is designed to copy the user input into a buffer of 8 bytes. However, there is no validation on the input size, and if the user provides input longer than 8 bytes, a buffer overflow can occur.

Buffer Overflow – Simple Example

Here's an attack scenario:

1. The attacker enters more than 8 bytes of data when prompted by the program.
2. Since the buffer is only 8 bytes, the extra data overflows into adjacent memory regions, possibly overwriting critical information, such as return addresses or function pointers.
3. By carefully crafting the input, the attacker may overwrite the return address of the copyData function with a memory address pointing to their malicious code.
4. When the function copyData finishes, the program returns to the altered return address, executing the attacker's code.
5. This allows the attacker to gain unauthorized access, control program flow, or execute arbitrary commands, potentially compromising the system's security.

Preventing Stack-based BoF Attacks

- The root cause does not come from OS but insecure programming practices.
- Programmers must be educated about the risks of insecurely coping user-supplied data into fixed size buffers.
- Use strncpy(buf, argv[1], sizeof(buf)) instead of strcpy(buf, argv[1]).
- **Function strcpy() copies the string in the second argument into the first argument**
  - e.g., strcpy(dest, src)
  - If source string > destination string, the overflow characters may occupy the memory space used by other variables
- **Function strncpy() copies the string by specifying the number n of characters to copy**
  - e.g., strncpy(buf, argv[1], sizeof(buf))
  - If source string is longer than the destination string, the overflow characters are discarded automatically

Shellcode Injection

- An exploit takes control of attacked computer so injects code to “spawn a shell” or “shellcode”.
- A shellcode is:
  - Code assembled in the CPU’s native instruction set (e.g. x86 , x86-64, arm, sparc, risc, etc.).
  - Injected as a part of the buffer that is overflowed.
  - We inject the code directly into the buffer that we send for the attack.
  - A buffer containing shellcode is a “payload”.
  - More information: https://samsclass.info/127/proj/p3-lbuf1.htm.

Buffer Overflow Mitigation

- We know how a buffer overflow happens, but why does it happen?
- This problem could not occur in Java; it is a C problem
  - ➢In Java, objects are allocated dynamically on the heap (except ints, etc.).
  - ➢Also cannot do pointer arithmetic in Java.
  - ➢In C, however, you can declare things directly on the stack.

Buffer Overflow Mitigation

- Address space randomization (ASLR)
- Data execution prevention
- Structured exception handler overwrite protection (SEHOP)