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**

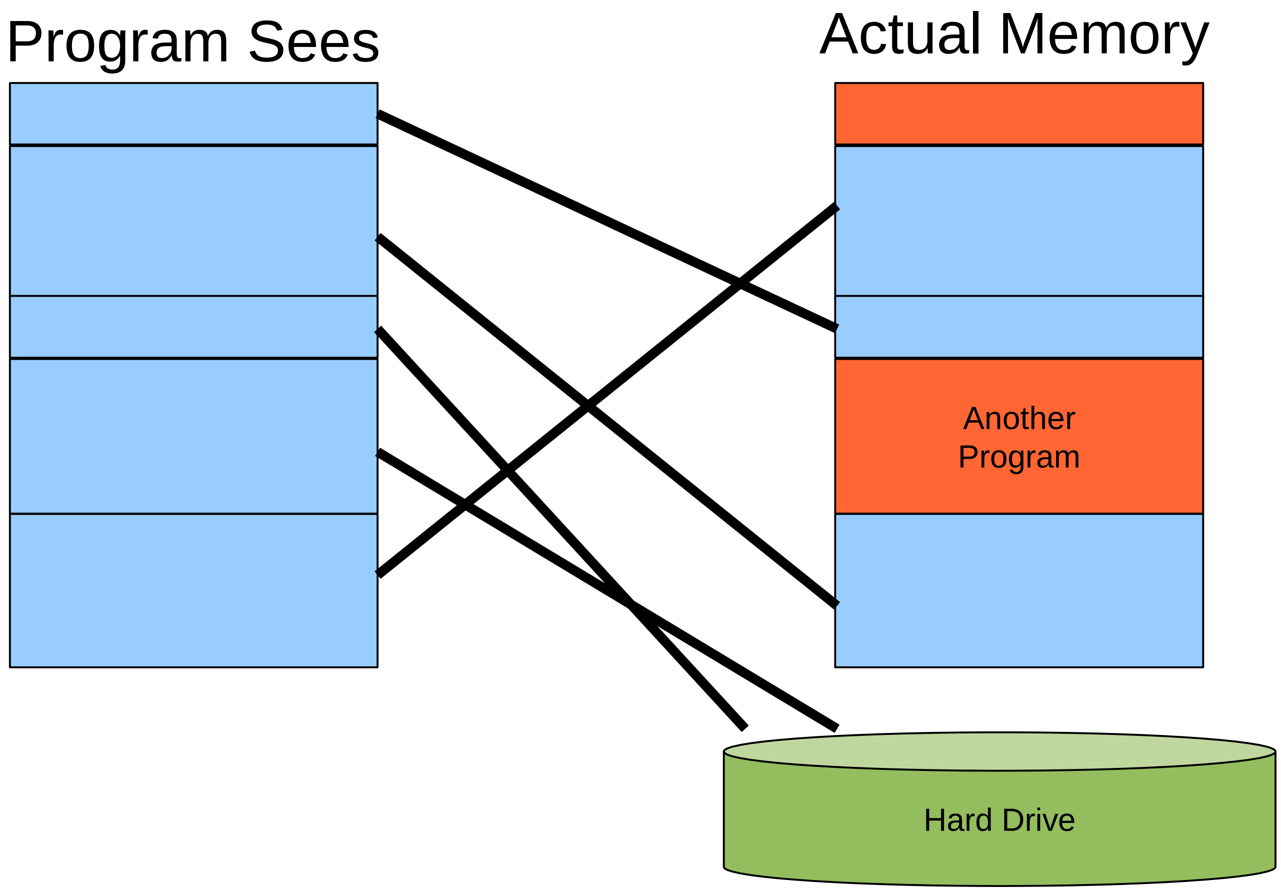
Buffer Overflow Attack

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

Address Space

- **Every program needs to access memory in order to run**
- **For simplicity sake, it would be nice to allow each process (i.e., each executing program) to act as if it owns all of memory**
- **The address space model is used to accomplish this**
- **Each process can allocate space anywhere it wants in memory**
- **Most kernels manage each process' allocation of memory through the virtual memory model**
- **How the memory is managed is irrelevant to the process**

Virtual Memory



Mapping virtual addresses to real addresses

Unix Address Space

- **Text: machine code of the program, compiled from the source code**

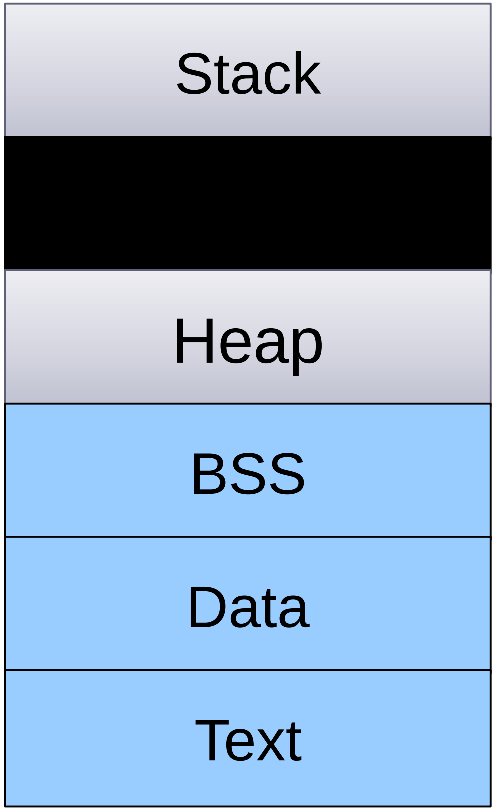
- **Data: static program variables initialized in the source code prior to execution**

- **BSS (block started by symbol): static variables that are uninitialized**

- **Heap : data dynamically generated during the execution of a process**

- **Stack: structure that grows downwards and keeps track of the activated method calls, their arguments and local variables**

High Addresses
0xFFFF FFFF



Low Addresses
0x0000 0000

Vulnerabilities and Attack Method

- **Vulnerability scenarios**
  - The program has root privileges (setuid) and is launched from a shell
  - The program is part of a web application
- **Typical attack method**
  1. Find vulnerability
  2. Reverse engineer the program
  3. Build the exploit

Buffer Overflow Attack in a Nutshell

- **First described in**

  Aleph One. Smashing The Stack For Fun And Profit. e-zine www.Phrack.org #49, 1996

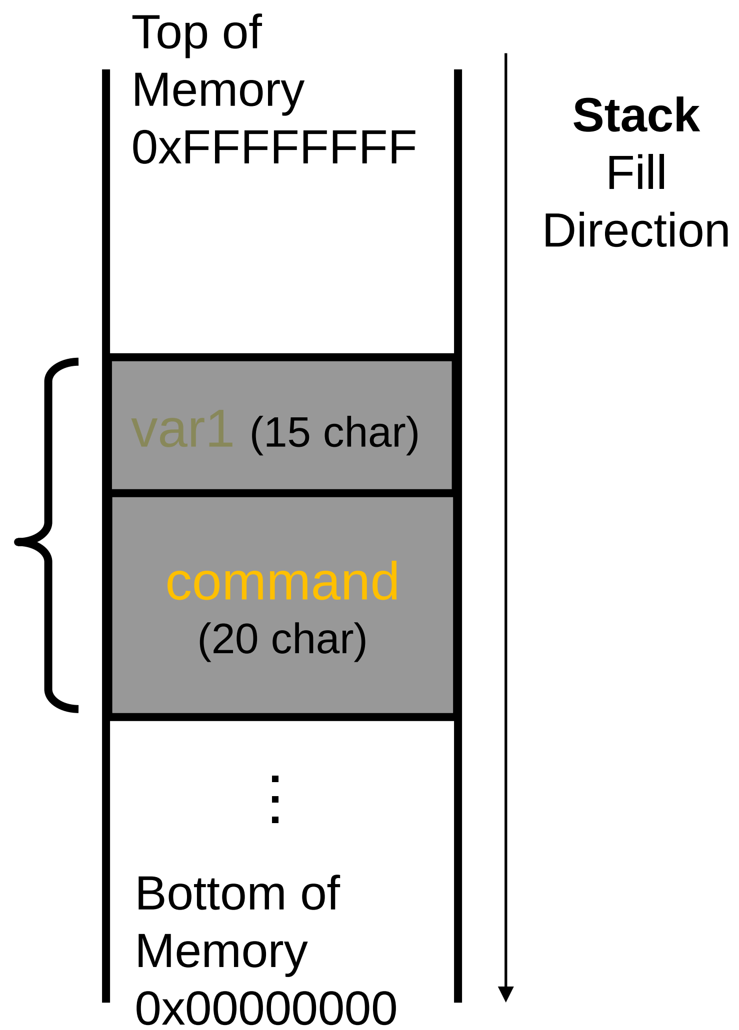
- **The attacker exploits an unchecked buffer to perform a buffer overflow attack**

- **The ultimate goal for the attacker is getting a shell that allows to execute arbitrary commands with high privileges**

- **Kinds of buffer overflow attacks:**
  - Heap smashing
  - Stack smashing

Buffer Overflow

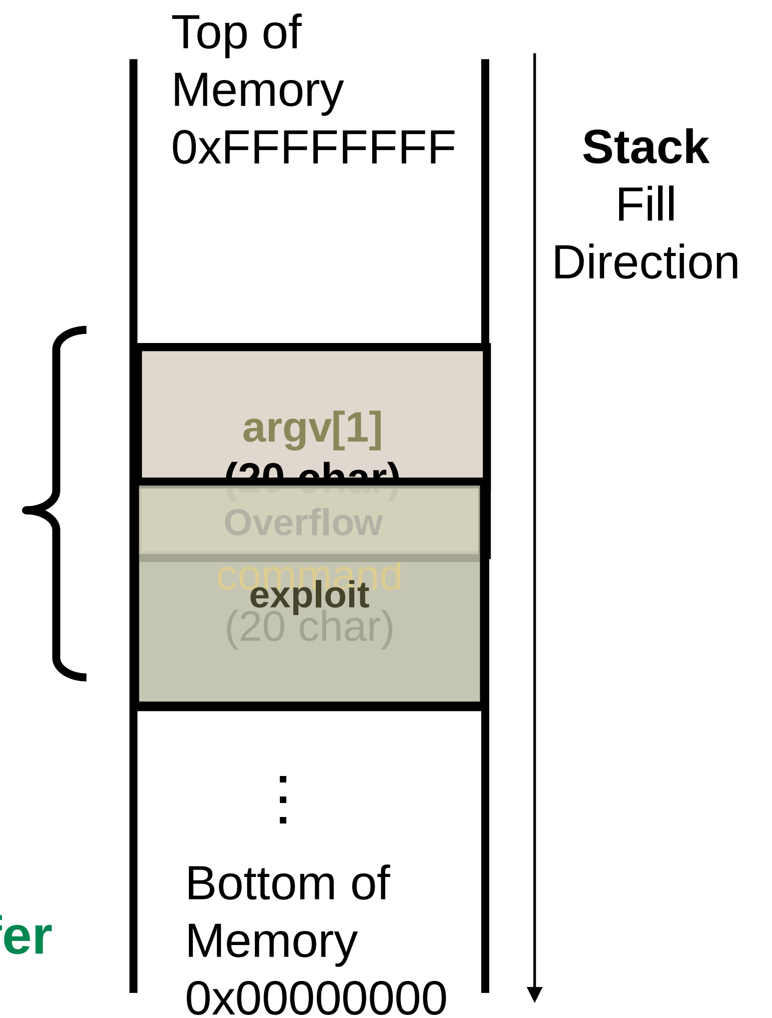
```
domain.c
Main(int argc, char *argv[ ])
/* get user_input */
{
    char var1[15];
    char command[20];
    strcpy(command, “whois ");
    strcat(command, argv[1]);
    strcpy(var1, argv[1]);
    printf(var1);
    system(command);
}
```



- **Retrieves domain registration info**
- **e.g., domain brown.edu**

strcpy() Vulnerability

```
domain.c
Main(int argc, char *argv[])
/*get user_input*/
{
    char var1[15];
    char command[20];
    strcpy(command, “whois ");
    strcat(command, argv[1]);
    strcpy(var1, argv[1]);
    printf(var1);
    system(command);
}
```



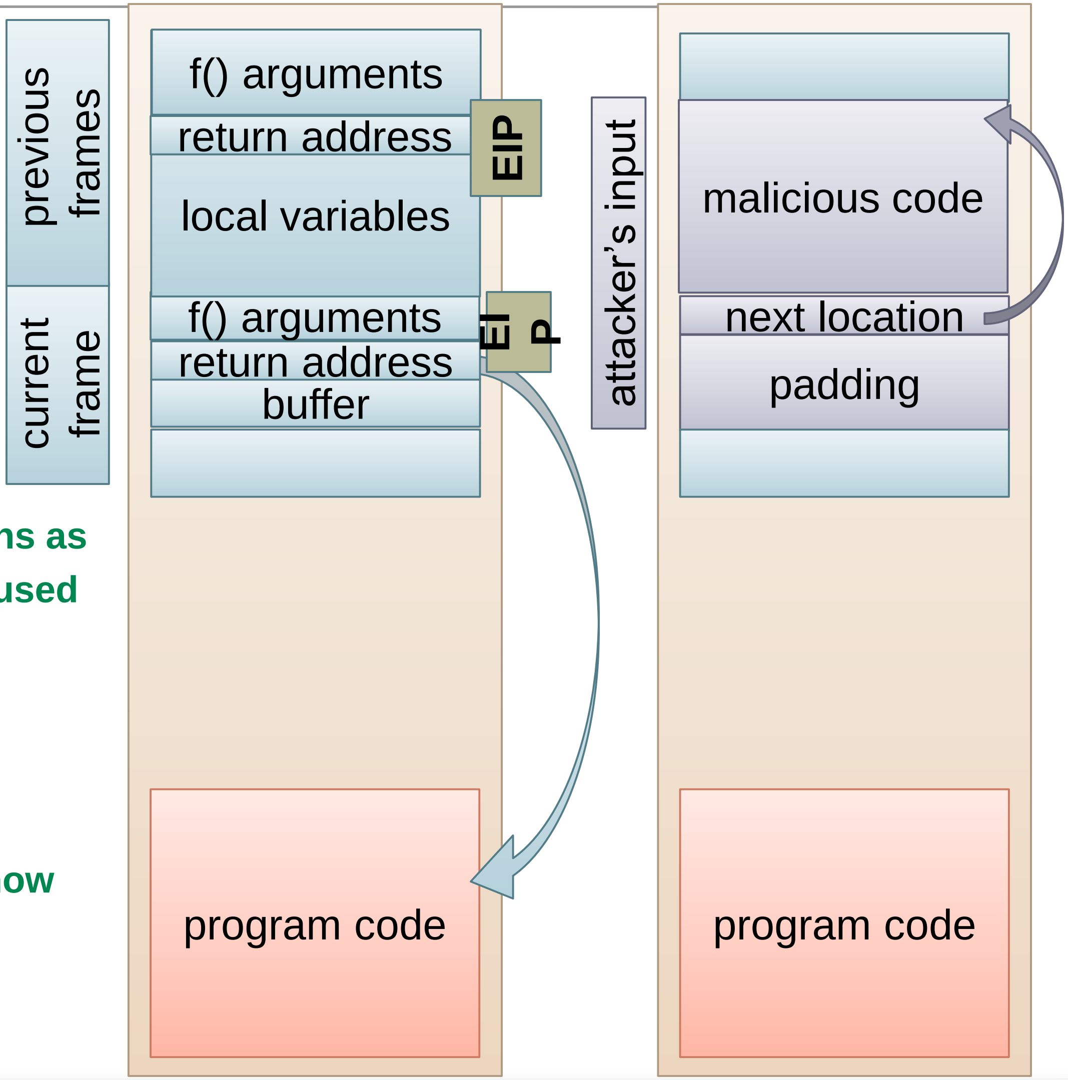
- **argv[1] is the user input**
- **strcpy(dest, src) does not check buffer**
- **strcat(d, s) concatenates strings**

strcpy() vs. strncpy()

- **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
  - The null character is appended at the end automatically
- **Function strncpy() copies the string by specifying the number n of characters to copy**
  - e.g., strncpy(dest, src, n); dest[n] = '\0'
  - If source string is longer than the destination string, the overflow characters are discarded automatically
  - You have to place the null character manually

Return Address Smashing

```
void fingerd (…) {
        char buf[80];
        …
        get(buf);
        …
}
```



- **The Unix fingerd() system call, which runs as root (it needs to access sensitive files), used to be vulnerable to buffer overflow**
- **Write malicious code into buffer and overwrite return address to point to the malicious code**
- **When return address is reached, it will now execute the malicious code with the full rights and privileges of root**

Unix Shell Command Substitution

- **The Unix shell enables a command argument to be obtained from the standard output of another**
- **This feature is called command substitution**
- **When parsing command line, the shell replaces the output of a command between back quotes with the output of the command**
- **Example:**
  - File name.txt contains string farasi
  - The following two commands are equivalent
  - finger `cat name.txt`
  - finger farasi

Shellcode Injection

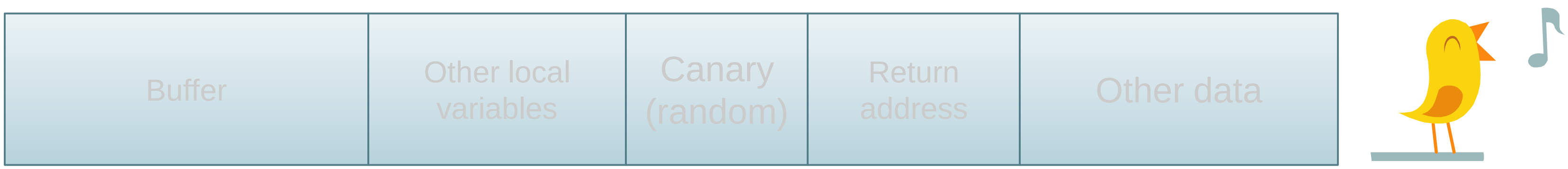
- **An exploit takes control of attacked computer so injects code to "spawn a shell" or "shellcode"**
- **A shellcode is:**
  - Code assembled in the CPU's native instruction set (e.g. x86 , x86-64, arm, sparc, risc, etc.)
  - Injected as a part of the buffer that is overflowed.
- **We inject the code directly into the buffer that we send for the attack**
- **A buffer containing shellcode is a "payload"**

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
- **One solution is to make the buffer dynamically allocated**
- **Another (OS) problem is that fingerd had to run as root**
  - Just get rid of fingerd's need for root access (solution eventually used)
  - The program needed access to a file that had sensitive information in it
  - A new world-readable file was created with the information required by fingerd

Stack-based buffer overflow detection using a random canary

Normal (safe) stack configuration:



Buffer overflow attack attempt:

- **The canary is placed in the stack prior to the return address, so that any attempt to over-write the return address also over-writes the canary.**