PEARSON NEW INTERNATIONAL EDITION

Introduction to Computer Security

Michael Goodrich Roberto Tamassia

First Edition

ALWAYS LEARNING

PEARSON

Pearson New International Edition

Introduction to Computer Security
Michael Goodrich Roberto Tamassia
First Edition

PEARSON

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  unsigned int connections = 0;
  // Insert network code here
  // ...
  // ...
  // Does nothing to check overflow conditions
  connections++;
  if(connections < 5)
    grant_access();
  else
    deny_access();
  return 1;
}
```

Code Fragment 3: A C program vulnerable to an arithmetic overflow.

An attacker could compromise the above system by making a huge number of connections until the connections counter overflows and wraps around to zero. At this point, the attacker will be authenticated to the system, which is clearly an undesirable outcome. To prevent these types of attacks, safe programming practices must be used to ensure that integers are not incremented or decremented indefinitely and that integer upper bounds or lower bounds are respected. An example of a safe version of the program above can be found in Code Fragment 4.

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  unsigned int connections = 0;
  // Insert network code here
  // ...
  // ...
  // Prevents overflow conditions
  if(connections < 5)
    connections++;
  if(connections < 5)
    grant_access();
  else
    deny_access();
   return 1;
}
```

Code Fragment 4: A variation of the program in Code Fragment 3, protected against arithmetic overflow.

4.3 Stack-Based Buffer Overflow

Another type of buffer overflow attack exploits the special structure of the memory stack. Recall from Section 1.4, that the stack is the component of the memory address space of a process that contains data associated with function (or method) calls. The stack consists of frames, each associated with an active call. A frame stores the local variables and arguments of the call and the return address for the parent call, i.e., the memory address where execution will resume once the current call terminates. At the base of the stack is the frame of the main() call. At the end of the stack is the frame of the currently running call. This organizational structure allows for the CPU to know where to return to when a method terminates, and it also automatically allocates and deallocates the space local variables require.

In a buffer overflow attack, an attacker provides input that the program blindly copies to a buffer that is smaller than the input. This commonly occurs with the use of unchecked C library functions, such as strcpy() and gets(), which copy user input without checking its length.

A buffer overflow involving a local variable can cause a program to overwrite memory beyond the buffer's allocated space in the stack, which can have dangerous consequences. An example of a program that has a stack buffer overflow vulnerability is shown in Code Fragment 5.

In a stack-based buffer overflow, an attacker could overwrite local variables adjacent in memory to the buffer, which could result in unexpected behavior. Consider an example where a local variable stores the name of a command that will be eventually executed by a call to system(). If a buffer adjacent to this variable is overflowed by a malicious user, that user could replace the original command with one of his or her choice, altering the execution of the program.

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  // Create a buffer on the stack
  char buf[256];
  // Does not check length of buffer before copying argument
  strcpy(buf,argv[1]);
  // Print the contents of the buffer
  printf("%s\n",buf);
  return 1;
}
```

Code Fragment 5: A C program vulnerable to a stack buffer overflow.

Although this example is somewhat contrived, buffer overflows are actually quite common (and dangerous). A buffer overflow attack is especially dangerous when the buffer is a local variable or argument within a stack frame, since the user's input may overwrite the return address and change the execution of the program. In a ***stack smashing*** attack, the attacker exploits a stack buffer vulnerability to inject malicious code into the stack and overwrite the return address of the current routine so that when it terminates, execution is passed to the attacker's malicious code instead of the calling routine. Thus, when this context switch occurs, the malicious code will be executed by the process on behalf of the attacker. An idealized version of a stack smashing attack, which assumes that the attacker knows the exact position of the return address, is illustrated in Figure 14.

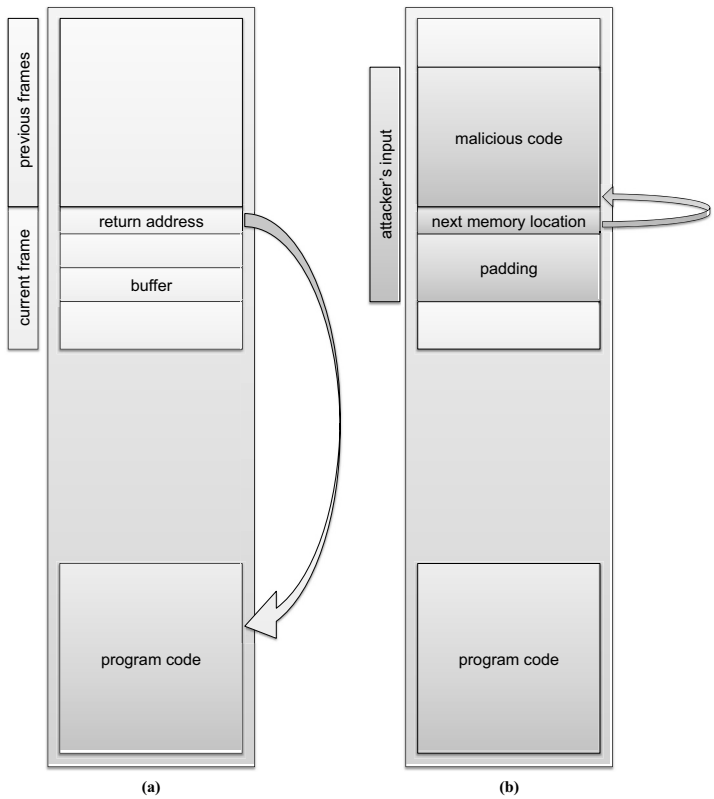

Figure 14: A stack smashing attack under the assumption that the attacker knows the position of the return address. (a) Before the attack, the return address points to a location in the program code. (b) Exploiting the unprotected buffer, the attacker injects into the address space input consisting of padding up to the return address location, a modified return address that points to the next memory location, and malicious code. After completing execution of the current routine, control is passed to the malicious code.

Seizing Control of Execution

In a realistic situation of a stack-based buffer overflow attack, the first problem for the attacker is to guess the location of the return address with respect to the buffer and to determine what address to use for overwriting the return address so that execution is passed to the attacker's code. The nature of operating system design makes this challenging for two reasons.

First, processes cannot access the address spaces of other processes, so the malicious code must reside within the address space of the exploited process. Because of this, the malicious code is often kept in the buffer itself, as an argument to the process provided when it is started, or in the user's shell environment, which is typically imported into the address space of processes.

Second, the address space of a given process is unpredictable and may change when a program is run on different machines. Since all programs on a given architecture start the stack at the same relative address for each process, it is simple to determine where the stack starts, but even with this knowledge, knowing exactly where the buffer resides on the stack is difficult and subject to guesswork.

Several techniques have been developed by attackers to overcome these challenges, including ***NOP sledding***, ***return-to-libc***, and the ***jump-to-register*** or ***trampolining*** techniques.

NOP Sledding

NOP sledding is a method that makes it more likely for the attacker to successfully guess the location of the code in memory by increasing the size of the target. A ***NOP*** or ***No-op*** is a CPU instruction that does not actually do anything except tell the processor to proceed to the next instruction. To use this technique, the attacker crafts a payload that contains an appropriate amount of data to overrun the buffer, a guess for a reasonable return address in the process's address space, a very large number of NOP instructions, and finally, the malicious code. When this payload is provided to a vulnerable program, it copies the payload into memory, overwriting the return address with the attacker's guess. In a successful attack, the process will jump to the guessed return address, which is likely to be somewhere in the high number of NOPs (known as the NOP sled). The processor will then "sled through" all of the NOPs until it finally reaches the malicious code, which will then be executed. NOP sledding is illustrated in Figure 15.

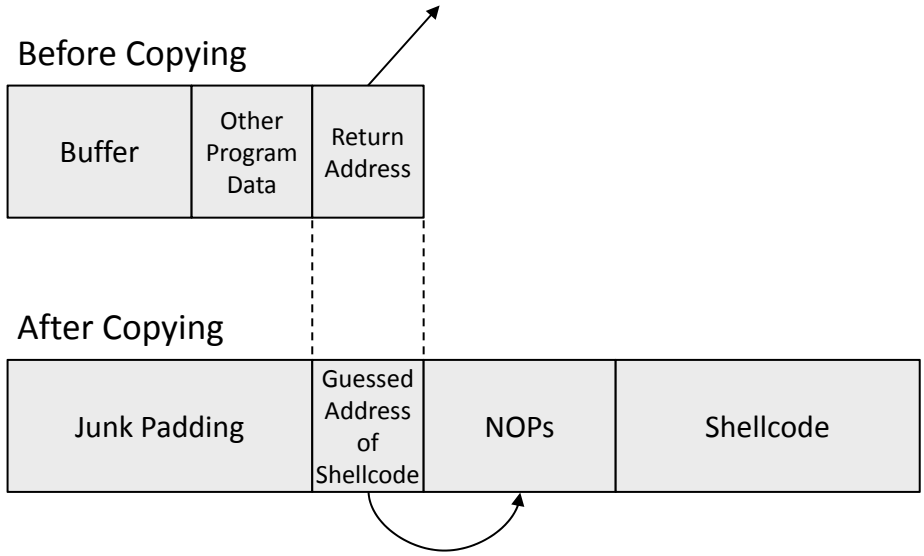

Figure 15: The NOP sledding technique for stack smashing attacks.

Trampolining

Despite the fact that NOP sledding makes stack-based buffer overflows much more likely to succeed, they still require a good deal of guesswork and are not extremely reliable. Another technique, known as ***jump-to-register*** or ***trampolining***, is considered more precise. As mentioned above, on initialization, most processes load the contents of external libraries into their address space. These external libraries contain instructions that are commonly used by many processes, system calls, and other low-level operating system code. Because they are loaded into the process's address space in a reserved section of memory, they are in predictable memory locations. Attackers can use knowledge of these external libraries to perform a trampolining attack. For example, an attacker might be aware of a particular assembly code instruction in a Windows core system DLL and suppose this instruction tells the processor to jump to the address stored in one of the processor's ***registers***, such as ESP. If the attacker can manage to place his malicious code at the address pointed to by ESP and then overwrite the return address of the current function with the address of this known instruction, then on returning, the application will jump and execute the `jmp esp` instruction, resulting in execution of the attacker's malicious code. Once again, specific examples will vary depending on the application and the chosen library instruction, but in general this technique provides a reliable way to exploit vulnerable applications that is not likely to change on subsequent attempts on different machines, provided all of the machines involved are running the same version of the operating system.

The Return-to-libc Attack

A final attack technique, known as a ***return-to-libc attack***, also uses the external libraries loaded at runtime—in this case, the functions of the C library, ***libc***. If the attacker can determine the address of a C library function within a vulnerable process's address space, such as system() or execv, this information can be used to force the program to call this function. The attacker can overflow the buffer as before, overwriting the return address with the address of the desired library function. Following this address, the attacker must provide a new address that the libc function will return to when it is finished execution (this may be a dummy address if it is not necessary for the chosen function to return), followed by addresses pointing to any arguments to that function. When the vulnerable stack frame returns, it will call the chosen function with the arguments provided, potentially giving full control to the attacker. This technique has the added advantage of not executing any code on the stack itself. The stack only contains arguments to existing functions, not actual shellcode. Therefore, this attack can be used even when the stack is marked as nonexecutable.

Shellcode

Once an attacker has crafted a stack-based buffer overflow exploit, they have the ability to execute arbitrary code on the machine. Attackers often choose to execute code that spawns a terminal or shell, allowing them to issue further commands. For this reason, the malicious code included in an exploit is often known as ***shellcode***. Since this code is executed directly on the stack by the CPU, it must be written in assembly language, low-level processor instructions, known as ***opcodes***, that vary by CPU architecture. Writing usable shellcode can be difficult. For example, ordinary assembly code may frequently contain the null character, 0x00. However, this code cannot be used in most buffer overflow exploits, because this character typically denotes the end of a string, which would prevent an attacker from successfully copying his payload into a vulnerable buffer; hence, shellcode attackers employ tricks to avoid null characters.

Buffer overflow attacks are commonly used as a means of privilege escalation by exploiting SetUID programs. Recall that a SetUID program can be executed by low-level users, but is allowed to perform actions on behalf of its owner, who may have higher permissions. If a SetUID program is vulnerable to a buffer overflow, then an attack might include shellcode that first executes the setuid() system call, and then spawns a shell. This would result in the attacker gaining a shell with the permissions of the exploited process's owner, and possibly allow for full system compromise.