Vulnerable SW - Buffer Overflow

Information Security – Lecture 24

Aadil Zia Khan

Executing Malware

- How can a malicious code inject itself into another program?
- How can a malicious code run with root privileges?

Buffer Overflow

Buffer overflow, or buffer overrun, is a situation where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites neighboring memory locations

How Is That A Problem?

- If Buffer Overflow overwrites data or executable code, it may result in erratic program behavior, memory access errors, incorrect results, and crashes

Stack Smashing

- A more serious threat caused by Buffer Overflow is Stack Smashing
- By sending in data designed to cause a buffer overflow, it is possible to write into areas known to hold executable code and replace it with malicious code
- If the affected program is running with special admin privileges, the malicious injected code would also have admin privileges
  - User can inject exec("/bin/shell") into a root program and gain root access on that shell

Taken from wiki

Exploiting Program Stack Buffer

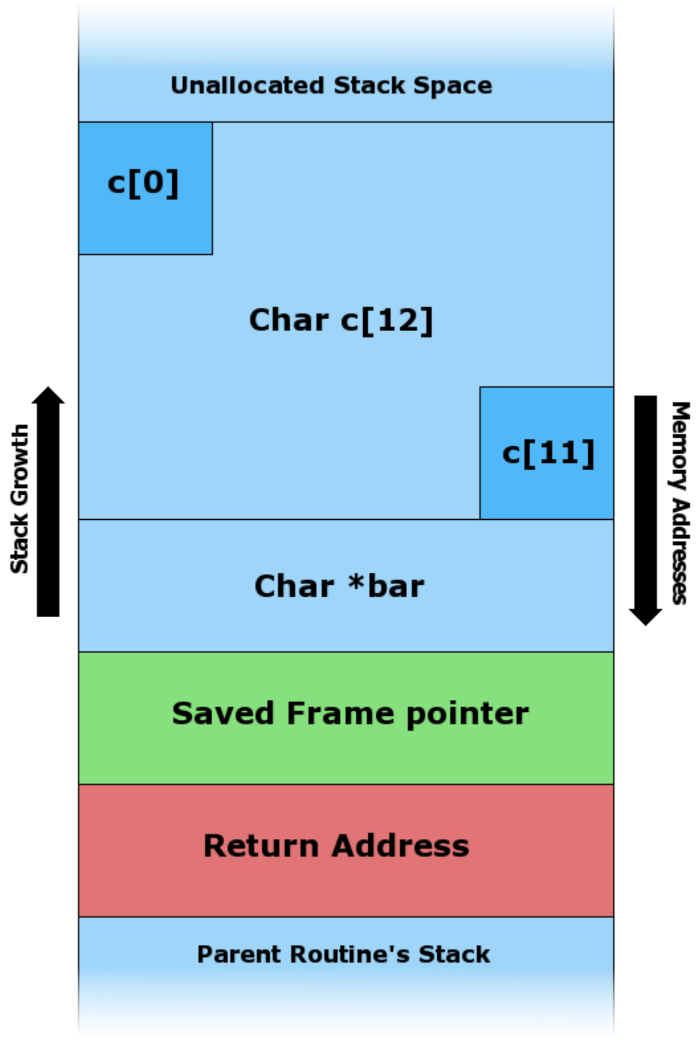
```c
//This code takes an argument from the command line and copies it to a local stack variable named c

#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;
}
```

Stack of foo function before data is copied.



Taken from wiki

Exploiting Program Stack Buffer

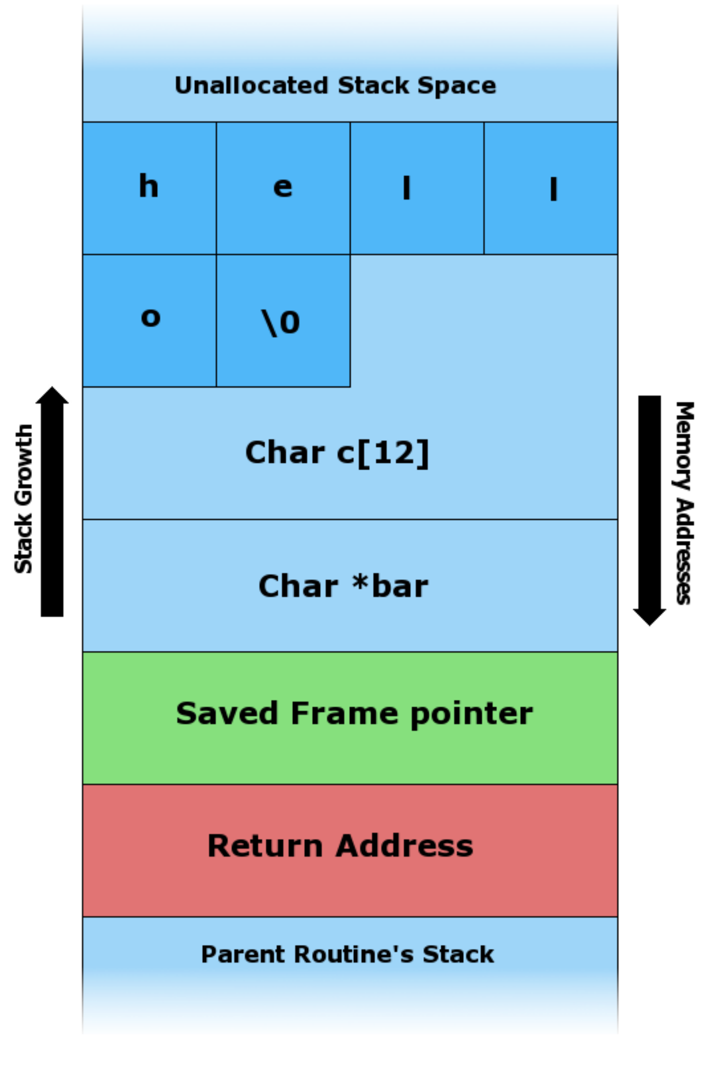
//This code takes an argument from the command line and copies it to a local stack variable named **c**

```
#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;
}
```

"hello" is the first command line argument.



Taken from wiki

Exploiting Program Stack Buffer

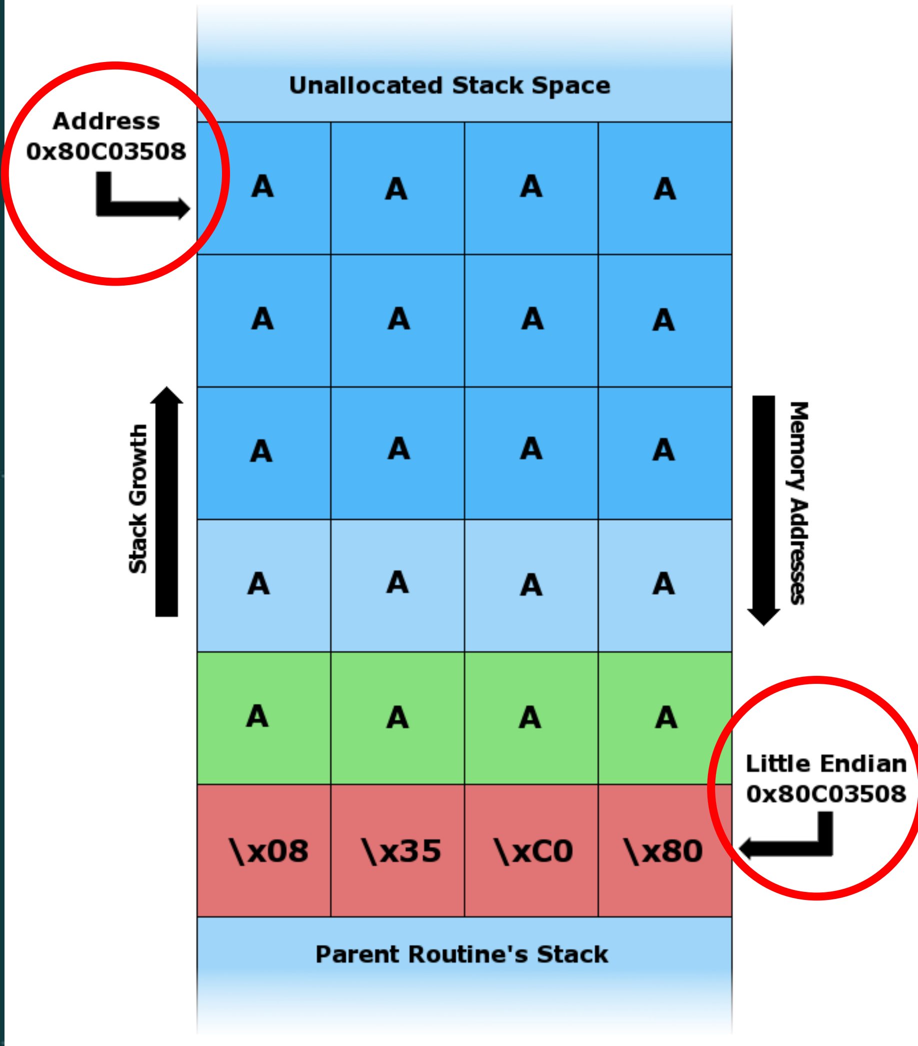
//This code takes an argument from the command line and copies it to a local stack variable named **c**

```
#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;
}
```

"AAAAAAAAAAAAAAAAAAAA\x08\x35\xC0\x80"
is the first command line argument.



Exploiting Program Stack Buffer Overflows

- When foo() returns it pops the return address off the stack and jumps to that address (i.e. starts executing instructions from that address)
- Note that the attacker has overwritten the return address with a pointer to the stack buffer char c[12], which now contains attacker-supplied data
- In an actual stack buffer overflow exploit the string of "A"'s would instead be code suitable to the platform and desired function
  - If this program had special privileges, then the attacker could use this vulnerability to gain superuser privileges on the affected machine

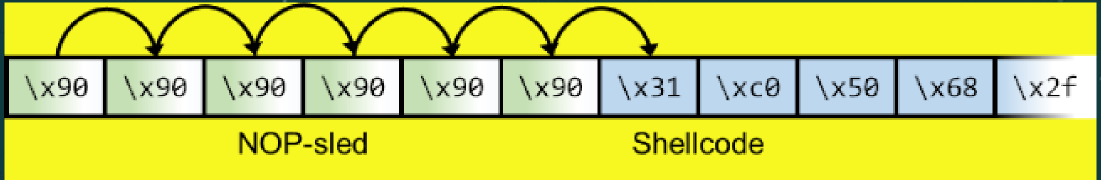
Complexity #1

- How do we copy a program to the stack buffer char c[12]?
  - Write a program and generate an executable file
  - Open this executable file using a disassembler
    - A disassembler is used to translate machine code into a human readable format
    - Get the hexadecimal values of the machine code
  - Copy these hexadecimal values into the buffer
    - Write ‘\x’ before each value to mark it as hexadecimal

This type of malicious code is also called the Shellcode

Complexity #2

- How do we know the starting address of stack buffer char c[12]?
  - We don't have to know the starting address - we use a NOP-sled
  - It solves the problem of finding the exact address of the buffer
  - A much larger section of the stack is corrupted with the no-op machine instruction
    - '\x90' in this case
    - No-op stands for no operation
  - This collection of no-ops is referred to as the "NOP-sled" because the execution will "slide" down the no-ops until it is reaches the actual malicious code by the jump at the end



Complexity #3

- How do we know the location where return address is stored?
  - We don't have to know the location
  - Similar to the No-Ops approach, we simply repeat the starting (approximate) stack address multiple times after the shell code so that it overwrites the location where return address is stored

One Shoe Fits All?

- Stack Buffer Overflow attacks depend on
  - Operating system
  - Growth direction of Stacks
  - Underlying hardware

Protective Countermeasures

- Use languages that do not allow direct memory access and do bounds checking
- Avoid standard library functions which are not bounds checked, such as gets, scanf and strcpy
    - Any program using these is automatically vulnerable and can be attacked
- Block packets which have the signature of a known attack, or if a long series of No-Operation instructions (known as a NOP-sled) is detected
- Randomization of the virtual memory addresses at which functions and variables can be found can make exploitation of a buffer overflow more difficult
- Stack canaries - a small random integer is placed in the memory just before the stack return pointer and checked to make sure it has not changed before a routine uses the return pointer on the stack

Famous Buffer Overflows

- The Morris worm in 1988 spread in part by exploiting a stack buffer overflow in the Unix finger server
- The Slammer worm in 2003 spread by exploiting a stack buffer overflow in Microsoft's SQL server
- Others include Witty worm, Blaster worm, etc.