C/C++ Buffer Overflows

A Presentation to the Unix Users Association of Southern California Orange County Chapter

Agenda

- Some of my background.
- What I hope you get out of the presentation.
- Background on computer architecture.
- Overflowing the stack.
- Overflowing the heap.
- Counter measures and counter-counter measures.
- How to avoid these mistakes.
- Resources for further information.

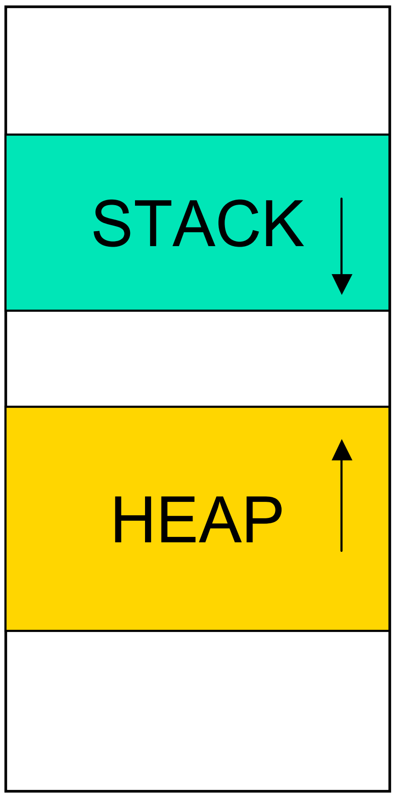
About Me

- Programming professionally since 1982
- Programmed in assemblers, Pascal, C, shell scripts, Java, *et.al.*
- Completed MSCS at LMU in December 2004 with original research into security issues of programming languages
- Currently working at USC Information Sciences Institute

What I hope you take away from the talk.

- Buffer overflows are a serious problem, *still*.
- Know the signs of the problem when working with legacy code.
- Know what traps to avoid when writing new code.
- Programming languages are tools and like any tool, can be mishandled.

Simplified Memory Architecture



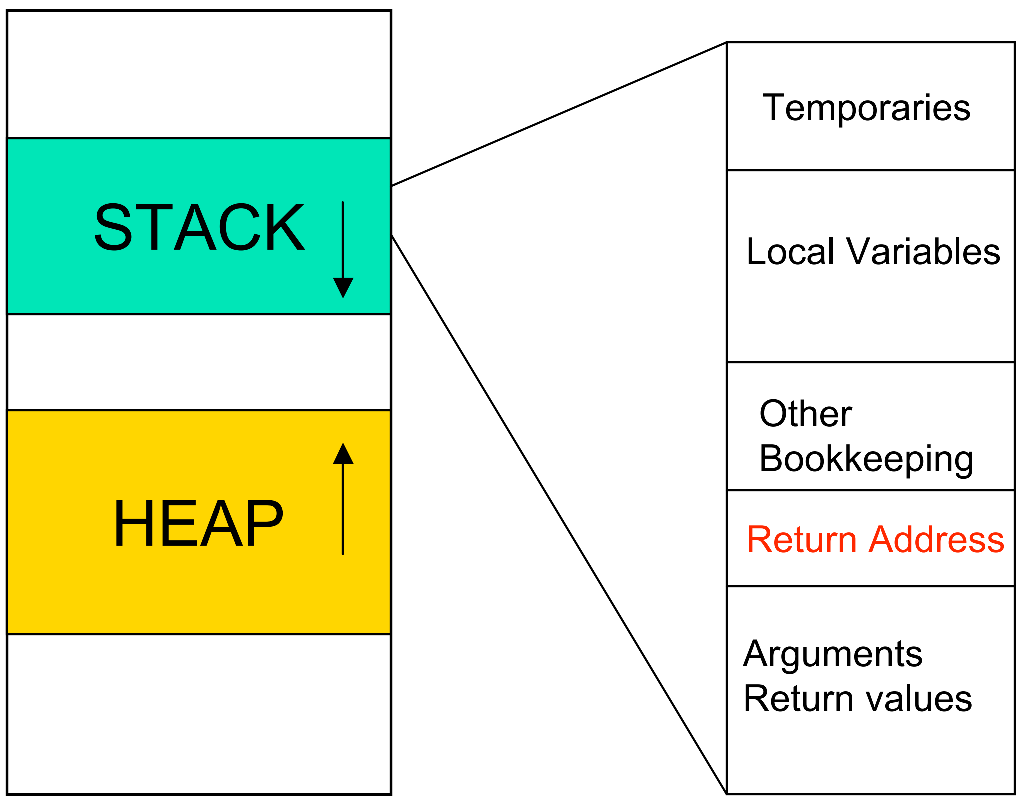
Higher Addresses

Stack grows down to lower addresses

Heap grows up to higher addresses

Lower Addresses

Simplified Memory Architecture



Basic Buffer Overflow: Stack Smashing

- Takes advantage of the lack of bounds checking in C and C++ languages.
- Uses an unchecked limit to inject alternate data into the stack.
- Usually used to inject “shell code” into the faulty program.

Basic Buffer Overflow: Stack Smashing

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

void function(char *str) {
   char buffer[16];
   printf("\nIn function...");
   strcpy(buffer,str);
   printf("Leaving function\n");
}

int main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';
  function(large_string);
  return(0);
}
```

Basic Buffer Overflow: Stack Smashing

```
$ ./example2

In function...Leaving function
Segmentation fault
$
```

Basic Buffer Overflow: Stack Smashing

- What happened?
  - The function allocated 16 bytes for the array.
  - The allocation was on the stack.
  - The caller sent 256 bytes.
  - These 256 bytes “overflowed” into other areas of the stack and “smashed” important booking data such as the proper return address to the function’s caller.

Basic Buffer Overflow: Stack Smashing

- Imagine what could happen if:
  - Instead of sloppy data, useable code had been used.
  - This code would effectively take over the flow of control of the program.
  - Often this is “shell code.”

Heap Buffer Overflows

- Heap storage is dynamically allocated at runtime.
- Rather than over-writing stack, these attacks over-write memory in the heap.
- Can be used to change values used for security processing.
- Often used with a stack smashing attack.

Heap Buffer Overflows

- As with stack smashing, the fundamental problem is that a program moves data into heap allocated memory without checking that the data will fit into the allocated space.

Heap Buffer Overflows

This example adapted from [Viega/McGraw, 2002] Example 7-12.

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

int main(int argc, char **argv)
{
        char            *str = (char *)malloc(sizeof(char) * 4);
        char            *super_user = (char *)malloc(sizeof(char) * 9);
        printf("Address of str is: %p\n", str);
        printf("Address of super_user is: %p\n", super_user);
        strcpy(super_user, "viega");
        if (argc > 1)
                strcpy(str, argv[1]);
        else
                strcpy(str, "xyz");

        printf("Value in buffer str: %s\nValue in buffer super_user: %s\n",
              str,super_user);
        return(0);
}
```

Heap Buffer Overflows

```
$ ./vm_example2A  abcd
Address of str is: 0x300140
Address of super_user is: 0x300150
Value in buffer str: abcd
Value in buffer super_user: viega
$ ./vm_example2A  abcdefghiklmnopqmcgraw
Address of str is: 0x300140
Address of super_user is: 0x300150
Value in buffer str: abcdefghiklmnopqmcgraw
Value in buffer super_user: mcgraw
$
```

Heap Buffer Overflows

- Imagine what could happen if:
  - The variable “super_user” was part of a security protocol: we have a new super user.
  - The platform didn’t allow executable code in the stack: Inject your shell code into the heap and change the return address to match your injected code.

Counter Measures

- StackGuard: “Canary” in the stack.
- StackShield: Return Address Copy.
- How each has been attacked: *Pointer Subterfuge*.

Counter Measure: StackGuard

- Uses an extra word, a "canary", in the stack.
- If the canary is changed, this signals that the stack has been smashed and the OS can take preventative measures, *e.g.* kill the program.

Counter Measure: StackGuard

- Two types of values are used for the canary:
  - The four bytes `0x000d0aff`
    - Most string operations terminate at NULL (0x00), carriage return (0x0d), linefeed (0x0a) or end-of-file (0xff)
  - A random number difficult to predict.

Counter Measure: StackShield

- StackShield makes a copy of the expected return address in another stack and this address is compared to the current return address when the function returns.

Counter-Counter Measures

- The techniques of StackGuard and StackShield both prevent many attacks, but imaginative people have found a way round them.
- Example from Bulba and Kil3r, “Bypassing StackGuard and StackShield”, *Phrack 56*, May 2000. (Article 5)

Pointer Subterfuge

```
int f (char ** argv)
{
        int pipa;// useless variable
        char *p;
        char a[30];

        p=a;

        printf ("p=%x\t -- before 1st strcpy\n",p);
        strcpy(p,argv[1]);        // <== vulnerable strcpy()
        printf ("p=%x\t -- after 1st  strcpy\n",p);
        strncpy(p,argv[2],16);
        printf("After second strcpy ;)\n");
}

main (int argc, char ** argv) {
        f(argv);
        execl("back_to_vul","",0);  //<-- The exec that fails
        printf("End of program\n");
}
```

Pointer Subterfuge

- Characteristics of a vulnerable program:
  1. A pointer such as *p* must be located next to a buffer such as *a*.
  2. A misused library routine that can execute an overflow into *p*. In the example, this is strcpy.
  3. A second copy function that uses *p* as the address of the buffer to write without *p* having been initialized.
- Authors identify one well-known program with these properties: wu-ftpd 2.5

GNU C++ VPTR Exploit

- Extremely narrow attack as it applies just to a particular implementation of C++ on a particular platform and not to the language in general.
- Builds upon the basic stack buffer overflow attack.
- Virtual Pointers, VPTRs, used to implement polymorphism in C++.
- Methods selected dynamically at runtime and not statically at build time.

Sample Code for VPTR Attack

```
class BaseClass
{
    private:
        char Buffer[32];
    public:
        void SetBuffer(char *String)
        {
            strcpy(Buffer,String);
        }
        virtual void PrintBuffer()
        {
            printf("%s\n",Buffer);
        }
};
```

```
class MyClass1:public BaseClass
{
    public:
        void PrintBuffer()
        {
            printf("MyClass1: ");
            BaseClass::PrintBuffer();
        }
};
```

This is just a snippet of the example.

GNU C++ VPTR Exploit

- The base class in the example defines a virtual function PrintBuffer. This requires derived classes to define their own version of PrintBuffer.
- GCC on IA32 stores pointer to these virtual functions on the stack.
- A stack overflow could, theoretically, over-write these addresses and change the copy of the method that is used.

GNU C++ VPTR Exploit

- This vulnerability is likely extremely difficult to exploit.
- It is important to note, however, because it shows that just because a language is object oriented does not mean that it is totally safe.

Common Thread

- Programs that use standard library routines that do not check bounds.
- Avoid using them. Many have a bounds checking counterpart, e.g. *strcpy* and *strncpy*.
- When you do use one, know why it's okay.

Static Code Analysis Tools

- There are tools available that are specifically designed to help find these problem areas before they can be exploited.
  - flawfinder
  - ITS4
  - findbugs
  - rats (Secure Software, Inc.)

Example run of flawfinder

```
$ flawfinder example2.c
Flawfinder version 1.26, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 158
Examining example2.c
example2.c:12:  [4] (buffer) strcpy:
  Does not check for buffer overflows when copying to destination.
  Consider using strncpy or strlcpy (warning, strncpy is easily misused).
example2.c:9:  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds checking,
  use functions that limit length, or ensure that the size is larger than
  the maximum possible length.
example2.c:17:  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds checking,
  use functions that limit length, or ensure that the size is larger than
  the maximum possible length.

Hits = 3
Lines analyzed = 27 in 0.76 seconds (102 lines/second)
Physical Source Lines of Code (SLOC) = 19
Hits@level = [0]   0 [1]   0 [2]   2 [3]   0 [4]   1 [5]   0
Hits@level+ = [0+]   3 [1+]   3 [2+]   3 [3+]   1 [4+]   1 [5+]   0
Hits/KSLOC@level+ = [0+] 157.895 [1+] 157.895 [2+] 157.895 [3+] 52.6316 [4+] 52.6316 [5+]   0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
```

Conclusions

- Accept that all programming languages have pluses and minuses.
- Know the assets and problems of your implementation language so you can balance speed, security, and maintainability.
- Regardless of language, define a coding standard that addresses security issues.
- Take advantage of the existing security guidelines for your language.

A Short Bibliography

- J. Viega and G. McGraw, *Building Secure Software*. Addison-Wesley. 2001.
- G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code*. Addison-Wesley, 2004
- R. Teer, "Secure C Programming." Sun Developer Network Community. http://developers.sun.com/solaris/articles/secure.html, 2001.
- J. Pincus and B. Baker. "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns", *IEEE Security & Privacy*, July/August 2004, pp. 20-27.
- Aleph One, "Smashing the Stack for Fun and Profit", *Phrack*, November 1996, http://www.phrack.org/show.php?p=49&a=14.
- D. Wheeler, *Flawfinder*, http://www.dwheeler.com/flawfinder/.

A Short Bibliography

- Bulba and Kil3r, "Bypassing StackGuard and StackShield", *Phrack*, May 2000, http://www.phrack.org/ show.php?p=56&a=5.
- J. Viega, J. Bloch, T. Kohno, and G. McGraw, "Token-Based Scanning of Source Code for Security Problems", *ACM Transactions on Information and System Security*, Vol. 5, No. 3, August 2002, Pages 238–261. (About development of ITS4.)
- D. Gilliam, J. Kelly, J. Powell, M. Bishop, "Reducing Software Security Risk through an Integrated Approach," *NASA Goddard Software Engineering Workshop*, pp. 36-42, November 2001.
- Rix, "Smashing C++ VPTRS", *Phrack*, May 2000, http://www.phrack.org/show.php?p=56&a=8.

Questions or Comments?