Interested in learning more about security?

SANS Institute InfoSec Reading Room

The Intrinsic Hole In Information Security

This discussion will address the lack of type safety as a fundamental weakness of the C program and how type safety coupled with the wide spread use of the C programming language relates to a massive hole in information security. The discussion begins with a historical perspective of the C programming language and why it is an integral part of so many computer systems. Then the discussion will cover type safety and how it relates to information security. Finally, the discussion wraps up with the safer alternative to C ...

Douglas Gaer
August 15, 2002
GSEC Practical Assignment Version 1.4
Option 1 - Research on Topics in Information Security

The Intrinsic Hole In Information Security

Introduction

Typically in the realm of information security topics such as social engineering, viruses, worms, password cracking, and the manipulation of the TPC/IP stack are the subjects of intense public scrutiny. There is an intrinsic area of information security that is does not receive the same level of public scrutiny due to the overall complexity of the issue. This is the issue of type safety in the C programming language. Time and time again buffer overflow attacks are used to compromise servers and workstations around the world. How does this happen? This discussion will address the lack of type safety as a fundamental weakness of the C program and how type safety coupled with the wide spread use of the C programming language relates to a massive hole in information security. The discussion begins with a historical perspective of the C programming language and why it is an integral part of so many computer systems. From there the discussion will cover type safety and how it relates to information security. Finally, the discussion wraps up with the safer alternative to C programming, C++ and some common methods used to make C programming more secure.

C Language Historical Perspective

The origin of the C programming language dates back to 1972 and was largely the work of an AT&T's Bell Laboratories systems engineer named Dennis M. Ritchie. In its beginning C language was developed primarily for writing the UNIX operating system. Although C started out as the language of operating systems it rapidly gained its popularity throughout the 1980s because it was well suited for developing programs on multiple operating systems. "In 1983, the American National Standards Institute (ANSI) established a committee to develop an official version of the C programming language [H2]. In 1988 the committee finalized a document defining the accepted standard for the language known as ANSI C [H2]." Almost every modern day C compiler now conforms to the ANSI standard or offers an ANSI compatibly mode. But long before the ANSI committee finished their work Bjarne Stroustrup, of the Computing Structures Research department at AT&T Bell Laboratories, begin working on an enhanced version of the C programming language first known as C with classes and later named C++ in 1983 [H9]. In fact several components of the ANSI C standard was adopted from C++ programming language.

What Is C Language?

C language is a procedural programming language where data is operated on directly. Technically speaking C language can be classified as a midlevel programming language that bridges the gap between hard to use assembly language and user friendly high level languages. Assembly language is a group of mnemonics that represent binary or machine code instructions specific to a particular microprocessor or a family of microprocessors. Programming in assembly requires an in-depth understanding of the computer system your working with, it's not portable, and can be extremely dangerous when used without years of experience and/or exceptional programming skills. High level languages such as Java, Pascal, Fortran, BASIC, etc are easy to learn, do not require an in-depth understanding of computer systems, and do not require years of experience and/or exceptional programming skills. What you gain from assembly is speed and some programs such as some device drivers can only be written in assembly. What you lose in high level languages is speed plus the use of low-level operations used to manipulate memory directly and to work directly with the CPU. C language rapidly became the most preferred alternative to bridge this gap leading to an industry wide explosion of source code that is still with us today. In C you have a relatively short learning curve (6 months to a year), the comparative speed of assembly language, the use of low-level operations, and high level construction built around high level control statements.

Composition of a C Program

A C program is simply a group of functions used to perform a programming task by manipulating data directly. Data in C programs is handled in the form of built-in data types (characters, integers, and floating points) and user defined data types, which are a grouping of built-in types within a data structure. By definition a C function is an independent collection of declarations and statements. However, when discussing C language it is important to think of functions as small standalone applications and not as subroutines that simply transfer control of the program from one part to another. Through the use of functions C programming allows for a modular implementation of source code, which enables the software developer to localize the code that manipulates the data. If implemented correctly the same code can be reused within the program or other programs. Localizing the source code into individual modules allows the software developer to make changes to the program faster and more efficiently. Modular implementation also makes it much easier to maintain the source code.

Portability In C

By design C language was initially developed as a portable language allowing programs to be compiled on many different operating systems. As part of the design strategy for making C portable, I/O (Input/Output) operations were placed outside the central definition of the language. "Most programming languages have I/O commands that are an inherent part of the language, but in C, the I/O operations are preformed by calling separate functions located in the a standard

library [H2].” The standard C library or LIBC is available on almost all modern day C compilers. As separate functions, I/O operations and other library components can be used by software developers to build executable programs on numerous computer systems independently of the underlying operating system. One fundamental issue to be aware of is that operations preformed inside LIBC are unique to specific compilers and operations systems. This is important when it comes to information security because a vulnerability that exists in one version of LIBC may not exist in another. Generally every operating system has a native compiler meaning a compiler produced by the operating system vendor. What complicates this is issue is that there are third party compilers available and since C is portable the third party compilers can be used in place of the native compilers. This means that a program compiled for the same operating system may or may not have a vulnerability depending on which C compiler was used to build the program.

C Data Types

Data in C is classified by type meaning the type of data your actually working with inside a function. As with any programming language C offers a rich set of built-in data types. The built-in types include characters, which are human readable letters, numbers, punctuation, and other symbols. Integer types are used to represent fixed size whole numbers and floating points types are used to represent whole numbers with a fractional part. In order to logically group data together user-defined types can be constructed using C structures and unions. User-defined types in C are constructed using built-in data types and other user defined types. Since C programs can manipulate memory directly a special built-in data type known as a pointer is used to hold the address of a memory location. Pointer types can hold the memory address of built-in data types, user defined types, or the address of a function referred to as a function pointer. Pointers are very powerful programming tools because they allow software developers to manipulate memory addresses directly. But if pointers are not used carefully they can become the source of a variety of type safety issues and run-time problems leading to unexpected program crashes, core dumps under UNIX, and the infamous “Blue Screen Of Death” under Windows. In C language a misused pointer is known as a “wild pointer” and wild pointers can be incredibly difficult to troubleshoot. Attackers often times look for function pointers in source code and running programs as a mechanism used to take control of a system by executing rogue machine language code. Function pointers are frequently used in many C programs because C language does not allow you to encapsulate functions in user-defined types.

Type Safety

Type safety is a set of rules enforced by the compiler to ensure that the correct data types are being operated on and used correctly. Built-in types, pointers, and user-defined types can be used in a C program as arrays (contiguous memory

buffers), function arguments, function return values, variables, or constants. C is not considered to be a type safe language because the compiler does not always force a software developer to use the correct data type, the compiler does not always ensure that the correct data is being used and the data type is being used correctly. In C, software developers can define names for variables and constants without associating a data type and can inadvertently use a data type incorrectly. In some cases the C compiler will automatically convert variables or constants of different data types in order to evaluate expressions consistently, referred to as implicit type casting. In C, software developers can also explicitly type cast one data type to another. What does all this mean? By not enforcing strong type safety it is possible to pass the wrong data type, return the wrong data type, or use a data type incorrectly. Variables and constants might be cast to the wrong data type by the compiler if the software developer does not use explicit type casts. Conceptually the use of the C macro and variable argument list facilities are not type safe practices at all. The most often exploited type safe issue in C is unchecked arrays. Unchecked arrays are the result of the compiler not checking the bounds of an array automatically and functions that do not check the bounds of an array. However, the use of unchecked arrays is not an oversight or something haphazardly implemented. Unchecked arrays are used for the purposes of speed and efficiency. But this practice can be dangerous and have a far-reaching affect when it comes to security. For example unchecked arrays are used throughout LIBC. So theoretically any direct use of “dangerous” LIBC functions could open a program to buffer overflow exploits and this means practically every C program.

Macros in C are used to define names for variables and constants. Parameterized macros can be defined to take arguments and call functions. The problem with macros in C is that they do not associate any data type with names or arguments. For example consider the following example macros:

```
#define end_of_doc "END"
#define print_string(a) printf("%s", a);
```

The first macro is used to define an end of document string and the second is used to create a short cut to print strings without repetitively typing the entire print statement. The type safety implication in both of these examples is that the compiler enforces no type checking at all. As you can see macros are time saving short cuts but it’s entirely up to the software developer to ensure that the correct data types are being operated on where a macro is used. Anywhere a macro is used it is possible to pass the wrong data type. Variable argument lists are another time saving short cut with type safety implications. The most common and well known use of a variable argument list in C is the formatted print function printf() declared as:

```
printf(const char *format, ...)
```

The ellipsis or… inside the printf() function tells the compiler that an unknown number of variables will be passed to this function. The format argument is used to insert strings, escapee sequences, and format data. An example of using the printf() function is as follows:

```
#include <stdio.h> /* Standard Input Output Functions */

int main(int argc, char **argv)
{
  printf("Hello World from program %s\n", argv[0]);
  return 0;
}
```

In this example the name of the program is passed to the printf() function as a character string and we tell the printf() function that argv[0] is a character string by using the %s format specifier. With this syntax the printf() function expects a single character string. What if the software developer uses the wrong format specifier, passes an integer type instead of string, or passes the wrong number of arguments? The C compiler will happily compile the program and produce an executable.

Unchecked arrays are a continual source of buffer overflow exploits and can be a programming nightmare for even experienced C language software developers. An array is a contiguous area of memory where a specified number of a single data type is stored. Contiguous memory means that all bytes allocated for the array are grouped together in a signal area of memory and all the array members are stored in order from lowest to highest starting with array member zero. Sometime arrays in C will be referred to as zero-based arrays as a reminder that array members start with member zero and not one. In C an array is declared by appending an array operator or the [] operator to a variable name, for example:

```
char  my_array[25];
```

This example declares a character array equal to 25 multiplied by the char data type size, which is equal to one byte or eight bits. Used in a program 25 bytes of contiguous memory will be reserved for the "my_array" variable, for example:

```
#include <stdio.h>  /* Standard Input Output Functions */
#include <string.h> /* Standard String Functions */

int main(int argc, char **argv)
{
  char  my_array[25];
  if(argc == 2) {
    strcpy(my_array, argv[1]);
    printf("%s\n", my_array);
  }
  return 0;
}
```

In this example an argument is passed to the program from the command line and copied into the “my_array” variable using the LIBC string copy function. This program will work fine as long as the string length of the argument is less than the array size minus one. In C your required to subtract one byte from a character array because the last byte is reserved for a NULL terminator used to signify the end of a string in memory. What happens when the argument exceeds 24 characters? A buffer overflow. Since arrays are contiguous the overage will spill into the area of memory starting after the last array member. Sometimes the program will crash and other times it will not depending on the amount of memory space overwritten. Ostensibly the problem could be corrected by using the LIBC string length function to check the length of the “argv[1]” variable before performing the string copy:

```
#include <stdio.h>  /* Standard Input Output Functions */
#include <string.h> /* Standard String Functions */

int main(int argc, char **argv)
{
  char  my_array[25];
  if(argc == 2) {
    if(strlen(argv[1]) < 25) {
      strcpy(my_array, argv[1]);
      printf("%s\n", my_array);
    }
  }
  return 0;
}
```

The problem here is that the string length function will overflow if the string is not properly terminated with a NULL character. Sometimes attackers will replace a NULL terminator in running a program causing string functions to overflow into other areas of memory. Another apparent fix would be to use LIBC strncpy() function but this function will also overflow if a null terminator is missing on the string being copied. NOTE: In this example the “my_array” variable is stored in stack space but could be stored in heap or free store space affecting other running programs well:

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

int main(int argc, char **argv)
{
  char  *my_array;
  my_array = malloc(25); /* Use heap space for the array */
  if(argc == 2) {
    strcpy(my_array, argv[1]);
    printf("%s\n", my_array);
  }
  free(my_array); /* Release the heap space back to the OS */
  return 0;
}
```

Buffer Overflow Attacks

As demonstrated in the "my_array" example code a buffer overflow is a condition that occurs when a program reads or writes pass the boundaries of a memory buffer, in this case a contiguous array. Once the bounds of an array has been violated attackers can inject code, manipulate function pointers, remove NULL terminators, and exploit other type safety weakness such as variable argument lists and macros. There are two kinds of well-known buffer overflow attacks: stack based and heap based. A new type of buffer overflow attack known as "Return-Into-Libc" is also starting to emerge [I15]. The stack refers to an area of memory assigned to a program to store local variables when the program is loaded into memory and during the invocation of functions within the program. The stack is sometimes referred to as scratch pad memory and can be thought of as a place where the program jots down information to remember variables and address locations. Heap space, also referred to as the "free store," is an area of memory available to applications to store data during a program's run-time. In C language heap space can be allocated and deallocated during run-time and is used to store dynamic data or data that will change in size during the life of a program. Data stored in stack space is static (fixed in length) and will not change in size during the life of a program.

A stack based buffer overflow is the most commonly used attack and is much easier to exploit than heap based buffer overflows or Return-Into-Libc buffer overflows. Since arrays stored in stack space are fixed in length attackers can easily calculate where an array ends and where the next variable or function address resides. When a program requests input from a local or network user an attacker can overflow the buffer by sending the program a string larger than the array can handle. If the program does not check the bounds of the array before storing the string in the array the buffer will overflow into other memory locations following the last byte of the array. This may cause the program to crash or allow the attacker to inject machine language code that the program will execute thinking it's the next function call after the array. Usually, the ultimate goal of code injection is to gain privileged access to the computer system giving the attacker full control of the system.

A heap based buffer overflow is harder to accomplish because the data is dynamic and constantly changing both in size and address location. A well-known heap based buffer overflow was the "Code Red" worm that exploited a vulnerability in the Microsoft IIS Web server [I2, I15]. The Return-Into-Libc exploit as documented on the Entercept Security Technologies Website tricks the program into calling LIBC functions and is currently a threat to UNIX systems [I15]:

http://www.entercept.com/ricochet/bufferoverflows.asp

Although writing a machine language program to gain privileged access sounds hard to accomplish, especially since every family of microprocessors have their

own machine language, it's much easier than you think. In fact attackers can produce small machine language programs with very little knowledge by using debugging tools provided by the compiler vendor. Debugging tools are command line and GUI based tools that allow software developers to troubleshoot compile time and run-time problems. Using a debugger an attacker can write a C language program that executes a shell or a shell command and dump the program in assembly language. The assembly language code can then be written in a hexadecimal format, which is a shorthand notation for binary. Debuggers can also be used to view stack variables and disassemble binary executables allowing the attacker to look for possible buffer overflow and code injection points. So don't think that open-source programs are more vulnerable than commercial programs distributed in binary form only. Once a program is released to the public in any form it's vulnerable. To emphasize this point visit the wtcracks.com Website [I21]:

http://www.wtcracks.com

Here you will find thousands of commercial software applications that have been hacked and can be transformed from time limited, key protected, and/or limited feature versions to full versions and released to the public for free. This is a case where hackers have decompiled and reverse engineered closed source software significantly jeopardizing the legitimate business profits of commercial enterprises. This also proves yet again that nothing on the public Internet is safe from attack.

Why Is Weak Type Safety In C A Hole In Information Security?

The lack of type safety in C language is a well-known cause of buffer overflows and there are other areas of information security to consider as well. A document published by Immnix research project describing the depth of the problem can be obtained from the following URL [I6]:

http://www.cse.ogi.edu/DISC/projects/immunix/discex00.pdf

According to this document: "All buffer overflow vulnerabilities result from the lack of type safety in C [I6]." The net affect of this fact is extremely serious when you consider that C language is the foundation of the UNIX and Windows operating systems. This means that every program running under UNIX or Windows will make a function call to a low-level C Application Programming Interface (API) no matter what language the program was written in. Not to mention the fact that billions of lines of C code have been written in the past 30 years and programs have a tendency to far out live their usefulness. When you consider that a vast majority of well-known and widely distributed network services are written in C it can easily be understood why type safety is an intrinsic hole in network security. And this problem extends beyond network security.

The lack of type safety in C continually jeopardize local security as well. For example the UNIX ping program can be used on UNIX systems to obtain privileged access to the system. How is this accomplished? In order for the ping program to work it must be ran with an effective user ID of root no matter what user you are logged in as. By causing the ping program to crash with a buffer overflow the ping program will drop the user into a root shell following the program crash. Most UNIX vendors have fixed known vulnerabilities to the ping program but how many more dangerous C programs are floating around? Can we rely on vendor patches to fix all vulnerabilities inside a program? The answer is definitely no. Often times the overall complexity of even the simplest program makes it incredibly hard to find violations of the typing system and possible code injection points. Plus often times programs will include debug modes and depreciated functions that are no longer used but stilling hanging around. So just because a vendor fixes known vulnerabilities inside a program that does not mean that there are not several more waiting to be discovered.

The Impact

This is an on going war that will never go away until all the C APIs have been rewritten in a type safe language or type safe wrappers are written for all the existing C APIs. What's the possibility of entire operating systems and large applications being rewritten industry wide? None, the ongoing trend is to keep patching the code that is readily available. A prime example demonstrating the impact of problem and the trend to keep patching is the latest OpenSSH buffer overflow vulnerability documented in the following CERT advisory [I3]:

http://www.cert.org/advisories/CA-2002-18.html

And UNIX is not the only target for buffer overflow exploits. The well known "Code Red" worm attack documented in the following CERT advisory used a buffer overflow exploit to compromise Microsoft IIS Web servers [I2]:

http://www.cert.org/advisories/CA-2001-19.html

The impact of the problem gets more and more severe when you look at the number of known buffer overflow exploits alone. For instance UNIX Remote Procedure Calls (RPCs) is the number one vulnerability to UNIX systems as documented in the SANS top 20 list [I16]:

http://www.sans.org/top20.htm

Also making the list at number five is the UNIX line printer daemon (LPD). Remote procedure calls are required to allow UNIX systems to share files using NFS and enable the use of the NIS system management tool. Every UNIX system that needs to print must have LPD running, which is usually every UNIX workstation and server attached to the network. As you can see the RPC and

LPD exploits alone can be of great significance to UNIX systems. The lack of type safety in C impacts network utilities as well. For example, the following CIAC information bulletin describes a remote buffer overflow vulnerability with version 3.5 of the TCPDump utility [I5]:

http://www.ciac.org/ciac/bulletins/l-015.shtml

To see several more known local and remote buffer overflows exploits visit the insecure.org Website at the following URL [I9]:

http://www.insecure.org/sploits.html

Here you will find a comprehensive list of buffer overflows exploits with detailed explanations including usage for a variety of operation systems: Linux, Solaris/SunOS, Microsoft, BSD, Macintosh, AIX, IRIX, ULTRIX/Digital UNIX, HP/UX, SCO, and others.

Why C Language Is Still Being Used

The commercial vendors, Universities, and individuals that produce open-source and commercial operating systems, network services, and applications chose to write their programs in C many years ago. Why did they do this when C++ was available as a more secure alternative and could have replaced C? The answer to this is simple. The continuing use of C language is a case of path dependency in an industry where massive change may lead to greater instability. The only way to fix the problem is to change everything at once and retrain everyone at the same time. But this is not likely to happen anytime soon and system administrators will be applying buffer overflow patches for a long time to come. And the use of C language is not going to diminish when you consider new products such as the Unified Parallel C Language and Compiler (UPC). UPC is a parallel extension of C language used with multiprocessors systems. More information about UPC can be found at the following URL [I12]:

http://www.super.org/upc/

The C++ Alternative

C++ is essentially a superset of C with object-oriented extensions [H6]. C++ is backward compatible with C but enforces strong type safety as well as many other features including replacements for macros, variable argument lists, and function pointers in user-defined types. Almost any C program can be compiled as a C++ program with a few changes to the source code. Some programs will compile with no changes at all. The downside to using C++ as a C compiler is that there may be some additional run-time overhead associated with your application. Unlike C, which is a procedural programming language, C++ is intended for use as an object-oriented programming language. But there is

nothing to prevent you from using C++ as a procedural programming language and taking advantage of the many new enhancements to the language including stronger type safety.

Object-oriented programming is entirely different programming concept with varying degrees of complexities. As with any object-oriented programming language C++ embodies the four major object-oriented programming concepts of abstraction, encapsulation, polymorphism, and inheritance. The central concept of object-oriented programming is that you no longer work on data directly. Objects, meaning an instance of a user-defined type, encapsulate all the functionality you need to manipulate the data. Given the complexities of object-oriented programming and the C++ language itself there is a steep learning curve associated with C++, typically two to three years. C++ also lacks standard APIs for various programming tasks such as multithreading, networking, database functionality, etc. This is why object-oriented languages such as Java far surpassed C++ in terms of popularity. However, Java is an interpreted language, which is costly in terms of speed and Java does not support multiple inheritance. Plus the Java interpreter still relies on the underlying C APIs to make system calls.

C++ is not an “out of the box” solution to the problem of type safety in C. The C++ compiler will not catch all violations of the typing system and allows the use of unsafe C programming methods such as marcos, variable augment lists, and unchecked arrays. Hence, all the dangers associated with C++ are allowed so that C++ will remain backward compatible with C. Although C++ is not an out of the box solution type safe wrappers can be build around the existing C APIs. This protects programs by providing a layer of type safety and prevents the misuse of API functions that can open a program to attacks and exploits. Well-written C++ wrappers will also simply complex programming tasks by encapsulating all the functionality of a C API into a single object. A truly object-oriented operating system and desktop would be a valuable asset to application development. Imagine an environment where programs could inherit functionality directly from the operating system and desktop.

What’s the Solution?

Unfortunately buffer overflows and other exploits of the C programming language are something we will have to live with for a very long time. The original foundations of many public information systems were not designed with security in mind. And given the complexities of modern day computer systems security still remains an event driven process for most system administrators and software developers. For those who suffered the disgrace of having their systems and/or programs compromised hard learned lessons will never be forgotten. And even though the hackers have the edge right now due to the share volume of work imposed upon system administrators and software developers they will eventually put themselves out of business. The more holes in information

security attackers expose the more secure computing will become. Secure programming may be a very obscure topic now but as the attacks continue secure programming will be the topic of more and more programming books, magazine articles, Web pages, and classroom discussions.

In reality the programs and operating systems used in business and industry today are a fact of life. It's the job of system administrators and software developers to keep the current systems working and secure as possible. C language has built the foundations of modern day computing and the use of existing C APIs will continue far into the future. But the very features that make C and C++ powerful programming languages can be the demise of many programs. In order to better defend programs against attacks C and C++ software developers have to always be conscious of information security and avoid unsafe programming techniques as much as possible. Cracking passwords and breaking encryption codes is just a matter of time. The same holds true for compiled and interpreted programs. Given some simple tools and enough time an attacker will find program vulnerabilities and exploit them. In closing here is some security conscious programming techniques in C/C++ gleaned from several Websites:

- Ensure that every new program is produced with security in mind.
- Remain overly cautious and aware that programs can be decompiled and reverse engineered.
- Never release code to the public compiled with any debugging options set.
- Treat all compiler warnings as errors and eliminate as many warnings as possible.
- Avoid the use of use of unsized arrays in C and C++.
- Where additional overhead is permissible always check the bounds of an array.
- Where additional overhead is permissible compile C programs as C++ programs.
- Use the type safe C++ Input/Output (I/O) stream libraries instead of the LIBC I/O library.
- Eliminate the use of define macros in all C++ code by using constants.
- Eliminate the use of parameterized macros in all C++ code by using inline functions.
- Use compile time polymorphic code in place of variable argument lists.
- Use C++ references in place of pointers to allow functions to modify arguments.
- Use the C++ new and delete operators over the C malloc() and free() functions.
- In C write type safe wrappers around the malloc() and free() functions.
- Use function pointers only when absolutely necessary.
- Write type safe wrappers around the standard C string functions.

- Use tools such as PC-lint [I10], FlexeLint [I10], and Purify [I14] to eliminate as many suspect lines of code as possible.

Internet References:

[I1] Bugtrag "Secure C Programming Summary." Online. Internet 14 Aug. 2002. Available <http://www.landfield.com/isn/mail-archive/1998/Dec/0107.html>.

[I2] CERT Coordination Center. "CERT Advisory CA-2001-19 "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL." Online. Internet 14 Aug. 2002. Available <http://www.cert.org/advisories/CA-2001-19.html>.

[I3] CERT Coordination Center. "CERT Advisory CA-2002-18 OpenSSH Vulnerabilities in Challenge Response Handling." Online. Internet 14 Aug. 2002. Available <http://www.cert.org/advisories/CA-2002-18.html>.

[I4] CERT Coordination Center. "Vulnerability Note VU#610291 Microsoft Internet Information Server (IIS) 4.0 and 5.0 buffer overflow in chunked encoding transfer mechanism for ASP." Online. Internet 14 Aug. 2002. Available <http://www.kb.cert.org/vuls/id/610291>.

[I5] Computer Incident Advisory Capability. "L-015: Tcpdump Remote Buffer Overflows." Online. Internet 14 Aug. 2002. Available <http://www.ciac.org/ciac/bulletins/l-015.shtml>.

[I6] Cowan, Crispin, et al. "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade." Online. Internet 14 Aug. 2002. Available <http://www.cse.ogi.edu/DISC/projects/immunix/discex00.pdf>.

[I7] Farrow, Rik "Blocking Buffer Overflow Attacks." Online. Internet 14 Aug. 2002. Available <http://www.networkmagazine.com/article/NMG20000511S0015>.

[I8] Frykholm, Niklas. "Countermeasures against Buffer Overflow Attacks." Online. Internet 14 Aug. 2002. Available <http://www.rsasecurity.com/rsalabs/technotes/buffer/buffer_overflow.html>.

[I9] Fyodor. "Exploit World" Online. Internet 14 Aug. 2002. Available <http://www.insecure.org/sploits.html>.

[I10] Gimple Software. "PC-Lint and FlexeLint." Online. Internet 14 Aug. 2002. Available <http://www.gimpel.com/html/lintinfo.htm>.

[I11] Harari, Eddie. "A Look at the Buffer-Overflow Hack." Online. Internet 14 Aug. 2002. Available <http://www.linuxjournal.com/article.php?sid=2902>.

[I12] IDA Center for Computing Sciences. “UPC: A Unified Parallel C Language and Compiler.” Online. Internet 14 Aug. 2002. Available <http://www.super.org/upc/>.

[I13] Kalev, Danny “Avoiding Buffer Overflows.” Online. Internet 14 Aug. 2002. Available <http://www.itworld.com/nl/lnx_sec/12182001>.

[I14] Rational The Development Software Company. “Rational Purify for UNIX.” Online. Internet 14 Aug. 2002. Available <http://www.rational.com/products/purify_unix/index.jsp>.

[I15] Ricochet Team. “Security Research.” Online. Internet 14 Aug. 2002. Available <http://www.entercept.com/ricochet/bufferoverflows.asp>.

[I16] SANS Institute Resources “The Twenty Most Critical Internet Security Vulnerabilities (Updated) The Experts’ Consensus.” Online. Internet 14 Aug. 2002. Available <http://www.sans.org/top20.htm>.

[I17] Teer, Rich “Secure C Programming.” Online. Internet 14 Aug. 2002. Available <http://soldc.sun.com/articles/secure.html>.

[I18] The FreeBSD Documentation Project. “FreeBSD Developers' Handbook Chapter 3 Secure Programming.” Online. Internet 14 Aug. 2002. Available <http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/x1216.html>.

[I19] Tsai,Timothy. “Buffer Overflow Vulnerabilities And Solutions”. Online. Internet 14 Aug. 2002. Available <http://www.icc2002.com/notes/ICC2002_TimothyTsai_BAS.pdf>.

[I20] Wheeler, David A. “Secure Programming for Linux and Unix HOWTO Chapter 5. Avoid Buffer Overflow.” Online. Internet 14 Aug. 2002. Available <http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/buffer-overflow.html>.

[I21] wtcracks Website. “Shareware Cracks.” Online. Internet 14 Aug. 2002. Available <http://www.wtcracks.com>.

[I22] Yossi. “Buffer overflows-getting started.” Online. Internet 14 Aug. 2002. Available <http://www.astalavista.com/library/auditing/bufferoverflow/gettingstarted.shtml>.

Hardcopy References:

[H1] Barkakati, Nabajyoti. Microsoft C Bible. Howard W. Sams and Company, 1988.

[H2] Himmel, David. Workout C. Waite Group Press, 1992.

[H3] Microsoft Corporation. Microsoft Visual C++ Run-Time Library Reference. Microsoft Press, 1995.

[H4] Microsoft Corporation. Microsoft Visual C/C++ Language Reference. Microsoft Press, 1995.

[H5] Oualline, Steve. Practical C++ Programming. O'Reilly & Associates, Inc., 1995.

[H6] Satir, Gregory and Doug Brown. C++ The Core Language. O'Reilly & Associates, Inc., 1995.

[H7] Schildt, Herbert. Teach Yourself C++. Osborne McGraw-Hill, 1992.

[H8] Schustack, Steve. C For Fun And Profit. Sams Publishing, 1993.

[H9] Stroustrup, Bjarne. The C++ Programming Language Second Edition. Addison Wesley, 1991.

Last Updated: November 18th, 2017

Upcoming SANS Training

Click Here for a full list of all Upcoming SANS Events by Location

SANS San Francisco Winter 2017	San Francisco, CAUS	Nov 27, 2017 - Dec 02, 2017	Live Event
SIEM & Tactical Analytics Summit & Training	Scottsdale, AZUS	Nov 28, 2017 - Dec 05, 2017	Live Event
SANS Khobar 2017	Khobar, SA	Dec 02, 2017 - Dec 07, 2017	Live Event
SANS Munich December 2017	Munich, DE	Dec 04, 2017 - Dec 09, 2017	Live Event
European Security Awareness Summit & Training 2017	London, GB	Dec 04, 2017 - Dec 07, 2017	Live Event
SANS Austin Winter 2017	Austin, TXUS	Dec 04, 2017 - Dec 09, 2017	Live Event
SANS Frankfurt 2017	Frankfurt, DE	Dec 11, 2017 - Dec 16, 2017	Live Event
SANS Bangalore 2017	Bangalore, IN	Dec 11, 2017 - Dec 16, 2017	Live Event
SANS Cyber Defense Initiative 2017	Washington, DCUS	Dec 12, 2017 - Dec 19, 2017	Live Event
SANS Security East 2018	New Orleans, LAUS	Jan 08, 2018 - Jan 13, 2018	Live Event
SANS SEC460: Enterprise Threat Beta	San Diego, CAUS	Jan 08, 2018 - Jan 13, 2018	Live Event
Northern VA Winter - Reston 2018	Reston, VAUS	Jan 15, 2018 - Jan 20, 2018	Live Event
SEC599: Defeat Advanced Adversaries	San Francisco, CAUS	Jan 15, 2018 - Jan 20, 2018	Live Event
SANS Amsterdam January 2018	Amsterdam, NL	Jan 15, 2018 - Jan 20, 2018	Live Event
SANS Dubai 2018	Dubai, AE	Jan 27, 2018 - Feb 01, 2018	Live Event
SANS Las Vegas 2018	Las Vegas, NVUS	Jan 28, 2018 - Feb 02, 2018	Live Event
Cyber Threat Intelligence Summit & Training 2018	Bethesda, MDUS	Jan 29, 2018 - Feb 05, 2018	Live Event
SANS Miami 2018	Miami, FLUS	Jan 29, 2018 - Feb 03, 2018	Live Event
SANS London February 2018	London, GB	Feb 05, 2018 - Feb 10, 2018	Live Event
SANS Scottsdale 2018	Scottsdale, AZUS	Feb 05, 2018 - Feb 10, 2018	Live Event
SANS Southern California- Anaheim 2018	Anaheim, CAUS	Feb 12, 2018 - Feb 17, 2018	Live Event
SANS Secure India 2018	Bangalore, IN	Feb 12, 2018 - Feb 17, 2018	Live Event
SANS London November 2017	OnlineGB	Nov 27, 2017 - Dec 02, 2017	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced