Spring 2021

Buffer Overflow and SQL Injection in C++

Noah Warren Kapley
Western Kentucky University, noah.kapley@gmail.com

BUFFER OVERFLOW AND SQL INJECTION IN C++

A Thesis
Presented to
The Faculty of the Department of Physics and Astronomy
Western Kentucky University
Bowling Green, Kentucky

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science

By
Noah Warren Kapley

May 2021

BUFFER OVERFLOW AND SQL INJECTION IN C++

Date Recommended April 22, 2021

Guangming Xing

Guangming Xing, Director of Thesis

Ivan Novikov

Ivan Novikov

Dominic Lanphier

Dominic Lanphier

Associate Provost for Research and Graduate Education

ACKNOWEDGEMENTS

Any challenging life-event such as embarking on an M.S. degree is wrought with pitfalls and opportunities. I would like to thank my parents, Dr. David and Trudy Kapley, for their unfailing and unyielding support of my pursuits and my sister, Sarah, for helping me with my bibliography, keeping me in good spirits, and making me aware of the importance of the “you-know-whos” of computer science and physics in my day-to-day world. Thank you, Dad, Mom, and Sarah.

My adviser, Prof. Guangming Xing, has bolstered my knowledge of computer security and computer science in innumerable ways. His cheerfulness, kindness, diligence, and dedication to his students shines forth in many different avenues on the fourth floor of College Heights Hall. I am honored to be his M.S. graduate student. Thank you, Professor Xing.

TABLE OF CONTENTS

BUFFER OVERFLOW AND SQL INJECTION IN C++

Noah Kapley May 2021 55 Pages

Directed by: Guangming Xing, Ivan Novikov, and Dominic Lanphier

Department of Physics and Astronomy Western Kentucky University

Buffer overflows and SQL Injection have plagued programmers for many years. A successful buffer overflow, innocuous or not, damages a computer's permanent memory. Safer buffer overflow programs are presented in this thesis for the C programs characterizing string concatenation, string copy, and format get string, a C program which takes input and output from a keyboard, in most cases. Safer string concatenation and string copy programs presented in this thesis require the programmer to specify the amount of storage space necessary for the program's execution. This safety mechanism is designed to help programmers avoid over specifying the amount of storage space in a computer in the event in which the actual storage space is smaller.

SQL injection into a computer database can alter or delete some or all of the computer database. To make matters more complicated, not all SQL databases use the same SQL statements and programming syntax. SQLite version 3 is a database which is vulnerable to SQL Injection. Computer Science I (CS I) and Computer Science II (CS II) classes will benefit from a computer program designed to illustrate various defective queries and how SQL injection might occur in a practical, real-world setting. The C++ command-line program designed in this thesis is a contribution to this project.

Chapter 1: Introduction

The progress of computer-related endeavors in today's technologically changing world presents society with the oft-neglected question of computer security. With convincing modern computer languages such as Java, .NET/CLR, JavaScript, PHP, and Ruby, why even worry about buffer overflows (BO), abbreviated as BoF in the academic literature, and SQL injection concerns (SQLI), especially when most enterprise applications seem to focus well under their corresponding web-frameworks of Spring or EJB, ASP.NET, Node.js, Zend, and Rails? Aren't these three issues merely a thing of the past, a cliché that "good programming" can solve in the now?

The complexity of applications in today's world leads programmers, even professional programmers, at times to err against security, sometimes with catastrophic consequences. C, a powerful, relatively low-level, and flexible systems language, is prone to BO when not correctly or safely used. The well-documented Morris worm (Tjaden, 2004, pp. 157-164) of the late 1980s is but one example of BO. In 2020, Equifax will pay $380.5 million to victims of the most disruptive SQL injection attack in history in response to hackers exploiting a 2017 deficiency in the customer service portion of the Equifax website.

Cybersecurity is necessary for today's online marketplace; cryptocurrency, banking, and shopping all require the careful and safe handling of computer systems. Moreover, cybersecurity is a national security issue, as domestic and international cybercriminals aim to disrupt online websites, government or otherwise, through malice and deception.

Education is one remedy for this situation. Towson University, for instance, offers cybersecurity-focused bachelor's and master's degree programs (Cybersecurity@Towson University, 2021) and many online modules aimed at education (Cybersecurity Modules: Security Injections| Cyber4All @Towson, 2021). These modules are aimed at providing classroom experiences for students enrolled in cybersecurity courses.

The development of workarounds and solutions to buffer overflow problems and demos for SQL injection studies will amplify and enhance cybersecurity curricula at the CS I and CS II levels. The first topic of this thesis regards the persistent buffer overflow vulnerability in C and C++ programs.

Chapter 2: Buffer Overflow

1. Introduction

To begin, a pronounced example of BO involves the Nintendo Wii game *The Legend of Zelda: Twilight Princess* in which the discovery of a buffer overflow, circa 2012, led to a proliferation of "user-designed" games when *Twilight Princess* was overwritten with too much computer information. In physics and nuclear engineering, C++ and Fortran command-line applications such as GEANT and MCNP must be designed so as not to admit BO. How many other C/C++ command-line applications might admit to BO, sometimes with fatal outcomes? This sliver of computer science security education is indeed important to the functioning of databases and console/command-line applications.

This BO section comprised of C programs investigates the need for safer, BO resistant solutions to common software problems. This BO tutorial contains programs that are short enough to run in online learning centers such as vLab, an interactive computer-programming website in which students may learn C++, Java, Python, and many other

concepts crucial to the development of computer-science skills. The author of this thesis (NWK) develops several C programs as alternatives to the fast yet unsafe functions in the standard C I/O and library prone to BO.

2. Literature Review

Diving into the literature on BOs in manuals on C and academic literature on cybersecurity categorizes salient research into a perspective that furthers the present motivations of this thesis. While buffer overflow "vulnerabilities have been hugely prevalent over the years and have been widely regarded as Public Enemy Number One that developers of such software need to avoid" (Stuttard & Pinto, 2011, p. 634), older manuals on C, in era where BOs were more of an issue in the development of C software, are mixed in their presentation of BO. Some older informal C language texts such as (Lafore, 1987) and (Waite, Prata, & Martin, C Primer Plus, 1989) refrain from discussing BOs altogether in regard to the standard C library functions gets, fgets, strcpy, and strcat—all of which can BO under the right circumstances. One older text aimed as a language reference for college students (Waite & Prata, C: Step-by-Step, 1990) and another reference manual for professionals (Harbison & Steele, 1991) do discuss the dangers of BO yet not by name. Additionally, a more recent text (Forouzan & Gilberg, 2001) informs students of the pitfalls of using functions in the C standard library.

The most seminal works on C (Kernighan & Ritchie, 1978) (Kernighan & Ritchie, 1988), through in their presentation of standard C library functions such as fgets, strcpy, and strcat, do not give further advice or workarounds for BOs. Graduate-level texts on computer security (Pfleeger & Pfleeger, 2007) (Du, 2017) (Tjaden, 2004) mention BOs but vary in their technical presentation of BO concepts and practical examples of BO in C.

While the software manuals and semi-popular texts either describe functions prone to BO or provide examples of BO, contemporary academic research into BO is wrought with attempts to cope with the persistent problem of BO. “Buffer overflow has been the most important vulnerability faced by developers” (Shaw, 2014) reads one article before delving into attempts to automatically transform some of the unsafe C functions noted in the last paragraph into program-safe alternatives immune to BO. The two methods of checking for BO in the literature involve static or dynamic schemas: static for the C code checking that occurs before the C program is compiled and dynamic for checking at the time at which the program is executed. Static programs for the grooming of C programs are stymied by several factors. One recent conference proceeding declares that a machine learning model to statically analyze C programs “do not perform that well when faced with larger, unlabelled data” (Zhao, Du, Krishnan, & Cristina, 2018). A static-analysis paper published near the writing of this thesis declared that the technique “results in a high number of false positives” (Shahab, Alenezi, Nadeem, & Asif, 2020). A Naval Postgraduate M.S. thesis (Wikman, 2020) outlines many facets of static BO analysis, including many contemporary programs that are used for static analysis. There is in these documents a need for a safer coding of standard C library functions prone to BO under improper use. It is this question which occupies this portion of the thesis.

3. Statement of Buffer Overflow Problem

A brief history of C at Bell Labs in the late 1970s by Thompson and Ritchie from the portable B assembly language is well-documented in (Kernighan & Ritchie, The C Programming Language, 1978, p. 2). While a knowledge of the concepts of assembly language is useful when seeing how necessary C was to the development of modern computer science, it is not necessary for viewing the problem of BO in C.

A buffer with finitely many contiguously-grouped free-storage or uninitialized elements x_1, x_2, x_3,…x_n is properly used if a C function does not access elements greater than x_n. A BO occurs when a C program proceeds to access and write on to elements x_m where m > n. Once the m > n overwriting condition is fulfilled, a BO occurs, and the program ceases to function normally. Some C programs will abruptly terminate while others will continue overwriting computer information that was never part of the contiguously designed buffer. If the overwriting persists, the computer hard drive will be noticeably altered.

BOs can occur with several functions in the standard C library. The functions strcat, "string concatenation;" strcpy, "string copy;" gets, "get string;" and fgets, "format get string" can produce buffer overflows over the right circumstances and overwrite computer memory. The development of safer functions in the C standard library begins in this thesis with strcat and addresses fgets in future sections.

4. String Concatenation in C

5. Introduction

At times, a C-String may need to be joined with another C-String. An example of this is in a banner displaying "Greetings, John!" after having prompted the user for his name. The program for strcat is taken from (Kernighan & Ritchie, 1988, p. 48). In the program, the user passes two char arrays to strcat. One array receives information while the other gives information. The net impact of strcat's normal operating instructions on the array receiving information becomes evident when the computer writes over the final

character in the receiving array called the null-terminator and then proceeds to “add” the contents to the end of the receiving array.

A dangerous problem arises when the receiving array is too small to hold the entirety of t; a pernicious BO occurs leading to undefined behavior. A summary example of strcat’s tendency to BO in a practical program is presented in Appendix A; the program works well for the name “Bose” but leads to BO for the name “Chandrasekhar,” named for two prominent Indian physicists of yesteryear. Another relevant example of strcat succumbing to BO implicates a vulnerable restaurant menu-receipt program. Appendix A buffer overflows if the user inputs “chicken,” “fried chicken,” or “fried turkey” if calling strcatSandwich().

While strncat (“string n concatenation”) is part of the standard library, copying only “n” characters from the C-String submitting information to the passive or “receiving” C-String, advocating another solution for this problem might prove to be fruitful for certain applications.

6. Solutions in C

6.1 Preliminary Non-ANSI Compliant Solutions

The discussion so far regarding strcat is that it is powerful, fast, low-level, and dangerous to use. While strncpy copies “n” characters from the submitting C-String to the receiving C-String, isolating the case in which the source C-String is too small for the receiving C-String will form the basis for the present investigation. Several of the C functions in the standard library are notorious for their ability to cause BO difficulties if improperly used by the developer. As a point of safety, requiring the programmer to specify

the size in bytes of the source C-String and receiving C-String will reduce errors and promote safe-coding practices.

The function sfstrcat is a safer version of Kernighan and Ritchie's "string concatenation." The code is presented in Appendix B. A momentary reflection about C programming yields the fact that the size of a buffer or C-String can always be provided to the developer, even within a function other than the main function. The function sfstrcat in its declaration takes as parameters a character array s, its size "lens," a char array t, and t's size "lent." The function strlen() found in (Kernighan & Ritchie, The C Programming Language, 1988, p. 99) returns the maximum number of whitespace (e.g. ' ' or '\n'), vacant (e.g. ''), and non-whitespace (e.g. 'a' or '1') entries in a null-terminated C-String before the strlen() encounters the null-terminator '\0'; in other words, the extra memory-space after '\0' will not be represented in strlen(). The expression "lens – strlen(s) – 2" ideally accounts for the "extra" space in the char array s; the final -2 term accounts for the null-terminator in char s [] and the null-terminator in char t [].

The interesting case in the if-statements is the "else if ((lens - strlen(s) - 2) < lent)" case. In this case, the "extra space" of char s []—the parent or receiving C-String—is smaller than the total space contained in the child or submitting C-String char t []. The initial lines of the "while" statement orient the counter "i" to be at the position of the null-terminator.

The declaration of "y" is critical to understanding the code; y is the "extra space" of char s []. Alternatively, the int y when ideally and properly used equals "the total length or number of bytes of s" minus "the space occupied by the C-String" minus "the space occupied by the two null-terminators." The -2 term is manifestly important for the

evolution of the program; in Kernighan and Ritchie's strcat, the null-terminator of char s [] is overwritten. A new null-terminator is placed at the end of the fully concatenated char s []. In sfstrcat(), the declaration of y is made before the null-terminator in s is overwritten. The while loop initialization statement includes a cryptic term to some; the postfix y-- term is an expression which decrements y after the truth-conditional y >= 0 is evaluated and before the expression s[i++] = t[j++] is evaluated. When y = 0, the "free space" in char s [] is almost used up; a null-terminator is placed at the end of the expression prior to a break in the loop.

The program sfstrcat can, of course, be used in any version of C provided the developer knows the total number of bytes in the receiving buffer char s [] and the total number of bytes in the "sending" buffer char t []. The reader may object that sfstrcat is not completely safe from BO. While this is true, the practice of the developer specifying the number of bytes in both the receiving and submitting C-Strings shall undoubtedly decrease the likelihood that a BO error will occur.

This form of sfstrcat is not, however, the most compact. The next section will further develop sfstrcat.

6.2 ANSI Compliant Improvements

The last section digressed into a typical method of treating sfstrcat. The present issue regards the possibility of improvement. Unlike sfstrcat(), the program under consideration does not assume that an unchecked strcat ought to be used at all in the design of the program. Rather, two checks are placed on the programmer: the entering of the maximum number of bytes in the receiving C-String and the entering of the maximum

number of bytes in the sending C-String, not too unlike strncat, which is in the standard C library. While these functions are slower than strcat(), their possibility for use in programs errs on the side of safe coding; the programmer in specifying both the submitting and receiving C-String parameters reduces the likelihood that an error can occur. This is enormous in large groups of programmers where one subset might write a test program as a demo using these "safe" functions while the other group may insert more "unsafe" functions in final release versions of the program.

The America National Standards Institute (ANSI) strncpy influenced the "Modified Safe String Concatenation" modsfstrcat program. Modsfstrcat is presented in entirety in Appendix C. The integer "spaces," or "space-s," is the "free" space in the char s [] including the null-terminator, which will be overwritten shortly. The null-terminator is included in the initial assessment of "spaces" for clarity, although the assignment statement in the first line of the program could be written as "spaces = lens – strlen(s) – 1." The null-terminator described in spaces for char s [] can be decremented in the assignment statement, in the first-if statement, or at any time before the second while-loop.

The integers "i" and "j" are used to obtain indices of the arrays s[] and t[]. The argument to the first if statement is designed to keep the developer from accidentally entering negative values into the arguments of modsfstrcat() or inserting values for lens and lent which lead to negative or zero values for the maximum byte-capacities reported by the developer for char s [] and char t [].

The maxt assignment statement involves C language constructs not yet covered in this thesis. The ternary-conditional operator (TCO), included in many C-Style languages

such as Java, JavaScript, PHP, or C++, is a convenient, short-handed way of writing an if-else statement explicitly and necessarily stated return values.

This talk of the TCO is useful in evaluating expressions in this short program. Using the TCO in arguments to iterative loops and, more commonly, as initializers for integers compacts programs. The threefold syntax for the TCO is: (truth-comparison statement) ? (statement if true or 1) : (statement if false or 0).

It is possible to observe that the TCO assignment of maxt incorporates thematic elements of strncat. The hints to program strncat covered in (Forouzan & Gilberg, 2001, pp. 547-548) and eluded to in (Kernighan & Ritchie, The C Programming Language, 1978) and (Kernighan & Ritchie, The C Programming Language, 1988) illustrate that strncat truncates the number of bytes in response to the choice of "n" or, in the vocabulary of the modsfstrcat, "maxt;" if maxt is greater than the sum of the maximum number of byte-characters plus the null-terminator, then maxt becomes the number of bytes contained in the C-String char t []. If the contrary is true in the TCO assignment, then maxt assumes the user-assigned value.

The while-loop described as "while (s[i]); i++;" ultimately orients the counter i to the position of the null-terminator '\0' of the char s [] array. The expression "while (s[i]);" employs a subtle technicality clearly explained in (Kernighan & Ritchie, The C Programming Language, 1978, p. 99):

> Since \0 is zero, and since while [loops test] only whether the expression [in the parenthesis] is zero, it is possible to omit the explicit test…

To reiterate, the expression "(s[i])" evaluates either to be some integer, as characters are integers in C, other than zero or zero itself—zero terminating the while-loop upon its evaluation.

The second while-loop overwrites in the C-String char s [] the null-terminator and the remaining number of bytes with the contents of the C-String char t [] under a few conditions. Unlike strcpy and strncpy, modsfstrcat terminates the iterations in the while-loop when spaces is equal to zero, that is, when the user input for lens subtracted from the "string length" of the C-String char s [] is equal to zero.

The prefix expression "--y" prevents a BO if the user faithfully enters a value of lens which is less than or equal to the number of bytes contained in char s []. Furthermore, the modsfstrcat program is safer than strncpy by requesting the user to supply the byte-size of both the receiving (i.e. char s []) and submitting (i.e. char t[]) character arrays.

As discussed previously, the integer "spaces" is the "total amount of free space" contained in the receiving character array char s []. The integer spaces, in a final estimation, becomes a limiter for the operations in the while-loop under current discussion. When y is decremented the first time, y is compared to zero. The result is that the "free byte-space" for the receiving character array char s [] ranges from y-1 to zero. For example, if there were 10 bytes of free space for the receiving character array char [] s, then the iteration in the while-loop ranges from 9, 8, 7…0. This is familiar to programmers used to zero-indexing for arrays.

As explained in many manuals on C, the postfix expressions "i++" and "j++" increment on the next line. The advantage to using the expression "(s[i++] = t[j++])" in the

program builds a culture of security into the program; the expression terminates the while-loop in the case in which the null-terminator for the submitting C-String occurs before the end of the receiving C-String. The assignment operator " = " in passing should be utilized in bracketing parentheses to preserve the order of operations.

With these introductory remarks in mind, modsfstrcat's rewriting in pointer-subtractive overtones shall evaluate low-level operations in a new manner.

6.3 Pointer Notation

As noted in the last section, it is beyond the scope of this thesis to fully define the relationship between pointers and arrays. Briefly put, though, pointers can hold addresses while arrays usually only hold data types. Pointers to structures, as shall be discussed in the fgets section of this thesis, need not be held in contiguous memory locations. Arrays, on the other hand, must always hold contiguous locations in memory-space and are vulnerable to stack BOs—the variant of BOs under discussion. A pointer-based program for string concatenation might be useful to developers as a succinct, compact alternative to array notation.

The "Pointer Safe String Concatenation" ptrstrcat program is expressed in Appendix D. Here, as in the previous section, the integer "spaces" is decremented accordingly and maxt is assigned through the TCO in the if-statement. The similarities, however, are limited beyond this observation. The character pointers "sstart" and "tstart" are reference values for the remainder of the program; as other pointers are incremented and decremented, sstart and tstart point to the beginning of the array in question. The first while-loop increments the pointer char *s to the null-terminator '\0' of the pointer array.

The second while-loop includes something which is new: pointer subtraction. This feature of C finds uses in the “String Length” strlen function in the “string.h” header file (Kernighan & Ritchie, The C Programming Language, 1988). The pointer subtraction term “s – sbeginCat” does not increment char *s. Rather, the term “s – sbeginCat” contributes to the argument of the while-loop the number of bytes between char *s and char *sbeginCat, the latter of which was declared before the initialization of the while-loop to serve as an electronic “placeholder” for the location at which the “free” space of the pointer array char *s contiguously begins. The “t – tstart” term also follows the same logic.

It is possible to equate arrays with pointers in ptrstrcat by passing to ptrstrcat either a character-pointer or a C-String for the proper functioning of the string-copy program. Furthermore, it is useful at this point to consult the order of operations for C in order to parse the meaning of the argument in the while statement. A glance at the statement *s++ = *t++ reveals that the incrementation of the pointer s or t occurs after the dereferencing and comparison of the two pointers. The argument of the while loop is somewhat troubling in an initial estimation. However, the last entry which is copied from t to s is the ‘\0’ null-terminator; once the null-terminator is copied in the while loop argument, the while loop terminates. This statement *s++ = *t++ is the basis for the “String Copy” strcpy function of the next section.

The “(*s++ = *t++)” term in this context is somewhat synonymous with the “(s[i++] = t[j++])” terms of previous sections of this thesis. As eluded to above, arrays and pointers are not the same. Arrays are, in a sense, an application of pointers. Pointers are generalized, as a Meijer-G function is to a Bessel function. Subsequent sections on pointers to structures will demonstrate the flexibility of the pointer approach to C.

In conclusion, if the ptrsfstrcat program were ill-constructed and well-used or well-constructed yet poorly used, incrementing a pointer or an array subscript would, again, lead the pointer into unknown territory, its dereferencing sometimes having disastrous consequences. It is impossible to implement bound checking for the char s [] of previous sections and the char * s of the current section at this computer-architectural level, as C does not allow users the option for implementing bounds checking on the arguments to functions themselves; for the case in which char * t contains more bytes than char * s, the user must faithfully insert a value for lens which is either less than or equal to than the sizeof(s), as the sizeof operator returns the number of bytes in s. A great amount of trust is placed in the programmer even when using these safer string functions. Groups of programmers must be careful and conscientious in their coding to avoid stack BO.

The function “String Copy” strcpy in the Standard C Library “string.h” is not immune to BO. The forthcoming examination of strcpy will further reinforce a Safer String Copy function.

7. String Copy in C

8. Introduction

There are times in the writing of a C program that the developer may wish to copy one string to another string. Passwords, e-mails, and other important quantities manifest themselves as strings in C programs. The program strcpy, “string copy”, is tersely written (Kernighan & Ritchie, 1988, p. 106). The pointer operations in the while-loop are explained in the previous section.

As an example of BO, the culprit string copy manifests itself well in a computer program designed to display a two-by-two matrix with each element on a new line. The program String Copy Matrix shown in Appendix E works well when the designer selects a value for the character array final[] of 9 bytes or more. However, a BO occurs when the developer selects an 8-byte value or fewer for the final character array. That this is the case is evident from counting the bytes in the original[] character array. The first three bytes in original[] are "1 1" while the "\n" newline character counts as one byte. The "2 2" in the original is 6-bytes long and the null-terminator '\0' is one byte. Altogether, the original[] array takes up 8 bytes worth of space. Eight bytes or less doesn't serve the program well. While also contrived, the code from (Miller & Zovi, 2009, p. 166) serves its purpose in illustrating the dangers of BO and strcpy. In Miller and Zovi's smashmystack, the argument-vector character-pointer array "argv" can accept any number of character-pointer entities from the user at the keyboard terminal. When smashmystack is called, a user may inundate argv with 1030-byte characters, for instance, and BO the program.

Like strcat, strcpy is fast, low-level, and dangerous. However, writing and implementing a function that requires knowledge of the number of bytes in the two buffers, as shall be seen, reinforces the security and good coding practices.

9. String Copy Solutions: Known Buffer Sizes

The function strncpy, which is part of the C standard library in the "string.h" header, can fall into misuse just as easily as strncat; the "n" bytes in strncpy might be greater than the submitting C-String. The result is chaos.

Requiring the developer to submit the size of the bytes of both C-Strings enforces safe-coding practices for copying strings. The function sfstrncpy, “Safe String N-Copy,” is useful if the programmer faithfully enters the integer size of the receiving C-String “d” into the function. Safe String N-Copy written in Appendix F employs checking mechanisms to offset the risks of BO. The integer “choices” employs the ternary-conditional operator; if lens is less than or equal to the length of the string + 1 to account for the null-terminator, then choices is equal to lens; if vice-versa. After its initialization, “choices” is used as a comparator for lend. In sfstrncpy, lend is the delimiter for lend <= choices; a buffer overflow, in this case, is avoided. The *d = ‘\0’ is necessary for the writing of the char* d array after the while loop’s termination.

While the above sfstrncpy function is useful for applications in which the user “faithfully” enters lend into the function, a “safe” yet “dangerous” pointer-subtraction form of strncpy, more applicable to strcat, is applicable to the BO problems at hand.

Format Get String or “fgets” is a C function that parses a C-String until a developer-specified number of characters is reached. Format Get String then returns control to the next function on the stack when the developer may choose to use the C-String result of fgets() operations. While fgets() is extensively used in C programs and is presented in its entirety in (Kernighan & Ritchie, The C Programming Language, 1978) and (Kernighan & Ritchie, The C Programming Language, 1988), there are problems with fgets() which are subject to present review.

10. Heap Input Parsing (HIP) Program

11. Problems with Format Get String: Ambiguities in C

The biggest problem in the formulation of fgets() is the occurrence of BO if fgets() has an int n value greater than the length of the pointer or array being passed to the function. Unlike in Java or C++, arrays in C do not automatically know their lengths. Rather, it is up to the programmer to specify what the length of an array is or is not. Another problem generic to C development is in the makeup of pointers and arrays; pointers are not the same as arrays, yet passing an array to fgets() in a C program automatically returns the first element—in C the "zeroth" element—of the array. The incrementation of the pointer, that is, arr += 1, yields the next element of the array. In fgets(), the function getch(FILE*) facilitates low-level input while the pointers increment throughout the array. Even if input is taken from getchar(), it is up to the programmer to specify a correct byte amount which is less than the total length—strlen + 1 for a null character—of the array. Any function which is designed to combat buffer overflow must yield to correct use by programmers; when using fgets, setting a byte limit value near the macro INT_MAX, for instance, is a recipe for disaster if the total number of characters to be read is ten bytes. One lengthy solution to this "Format Get String" problem is to propose a heap-based alternative to fgets. Advancing such an agenda will consume the space of several sections of this thesis.

12. Introduction to Heap Input Parsing (HIP)

The use of data-type constructs to control input parsing is nothing new; using "n" to record the number of bytes maximally encountered when a character pointer or array is submitted to Kernighan and Ritchie's famous fgets program is well-used in the legacy of C. It is difficult to improve upon fgets. All elements of the program are fast, low-level, and have the cybersecurity disadvantages previously discussed. Still, the author of this thesis (NWK) pondered what impact a Heap-Based Input Parsing Program (HIP), using dynamic-

memory allocation to test each character element of the standard-input stream, might have on the evolution of a C program. The concluding BO sections of this thesis aim to advance, develop, and analyze what such a HIP program would do for the C developer.

Although HIP is far slower by algorithmic standards than fgets, several points contrasting HIP to "Format Get String" will help elucidate the usefulness of HIP. The main difference between HIP and fgets is in the use of the "Get Character" function getc from the C Standard Library; HIP uses structures noncontiguous from one another to store information from standard input, even the command-line, whereas fgets uses pointer-arrays with elements contiguous "somewhere" in memory-space. When the standard input is emptied by HIP, a fully formed character array will return just as if fgets were used. While HIP, unlike fgets, relies almost exclusively on structures allocated from the heap, HIP is useful for applications lacking knowledge about the standard input stream, for instance, the allocation of a 2D array in C. A HIP program included in Appendix G illustrates the ordering of keyboard user input to the standard output computer monitor. Appendix G uses linked lists to process the information from getc.

The HIP's complicated code demands that an inquiry be made at each execution step of its constituent subprograms. The subsequent chapters covering the verbose C function will focus on Pointer error checking for dynamically allocated pointers (e.g. malloc), dynamic-memory allocation of HIP structures, the creation of linked-list data structures suitable for storing dynamically allocated nodes, the maintenance of linked-list data structures (viz the addition of nodes and additional structures to the linked-list), the transfer of the linked-list information to single or multiple arrays for further manipulation—arrays contiguous in memory-space are vulnerable to stack overflows (e.g.

char x [1000000000000] might not work on some computer architectures)—and the deletion of the linked-list and all nodes comprising the linked-list.

Pointer error checking, then, is the first order of business. The resulting error checking for HIP is simple and reusable. As stated above, the uninitialized pointer encountering malloc is assigned a NULL value when malloc's call to the operating system fails. The null pointer returned to the program, then, becomes an important puzzle piece. One form of error checking a NULL pointer regards the pointer as containing an address that cannot be zero. Examples of this found in (Daconta, 1993, pp. 65-69) shall find use in HIP. The integer, structure, and character pointer error checking functions for HIP are denoted in Appendix G: chkAllocChararr, chkAllocInt, and chkAllocChar.

A few remarks regarding the error checking mechanisms will lend some clarity to the evolving discussion of HIP. The bang-exclamation mark operator ! reverses the truth-functionality of the resulting expression; if "if(x > 5)" is true or any integer other than zero, then "if(!(x > 5))" states "if it is false that x > 5, then execute the loop." It is possible to envision comparing the statement "if(!ca)" with the statement "if(!(ca != NULL))". The if-statements in this section are only executed in the event in which a pointer is NULL. What happens when if the function chkAllocChar, for instance, executes the body of the if-statement? An error message is sent to the standard error stream, in many cases a computer monitor, and the program is abnormally terminated, hence the "exit(1);" line. For convenience, the error checking functions return to the program the dynamically allocated pointer in question.

For purposes of completeness, it is useful to consider the motivation behind using singly-linked lists for the development of HIP.

13. Singly Linked Lists in HIP

The sections regarding strcat and strcpy in this thesis regarded the development of C functions safer to use during development and debugging. These “String Concatenation and Copy” functions were terse, iterative, and lacked recursion. There are, however, other ways in C to accomplish I/O, namely the method of HIP.

The “Character Array” chararr struct that HIP uses for its processing of user input is near the beginning of Appendix G. Furthermore, HIP instantiates the chararr struct by using the C heap-memory allocator malloc and uses dynamic memory allocation at run-time. An example of this is present in Appendix G’s initchararrll() function. The sizeof operator is used to contribute the number of bytes of a struct chararr to the program. The parentheses around struct chararr* denotes pointer casting; the memory from malloc assumes a chararr structure pointer. At the time of this writing, some implementations of C do not require the “struct chararr*” casting on the left-hand side of the “struct chararr* c” definition (e.g. GNU C++) while another implementation of C++ does (viz Visual C++).

Singly-linked lists central to the development of HIP are used for their ability to avoid overflow until the computer is practically exhausted of memory (Kruse, Leung, & Tondo, 1991, pp. 120-121). The reader who sedulously views Appendix G in anticipation of the next section may object that the range of an integer is the limiter for this process. Nevertheless, contiguous memory at the time of this writing cannot be allocated with the number of blocks exceeding the maximum range of an integer. A generalization of HIP important for the processing of command-line arguments is yet another application for this “HIP” method of thinking.

14. Adding Linked List Nodes to the End HIP

HIP employs singly-linked lists for flexibility; standard input can submit plenteous amounts of information to a HIP-linked list. While a contiguous-memory array is faster in iteration and member access than a HIP-linked list, it is worth stating that a HIP-linked list's redundant systems are stable against BO attacks. While fgets is low-level and fast, HIP can redundantly process a user at a terminal in the absence of using an "n" byte delimiter.

The function "Add a Node and Return the Last Non-Null Node" is extensively used in HIP. The function found in Appendix G takes a struct chararr * pointer "head," iterates to the last non-null node, and uses initchararrll() to dynamically create a new node with heap memory. If head is equal to 0 or NULL, an error message displays, and the program is terminated. The else statement processes the main body of the function. If head is not equal to NULL, a while-loop checks whether head is anything other than NULL in the argument of its loop call. In the while loop, if head->next is NULL, if the next linked-list unit is at the end of the list, then initchararrll() dynamically initializes x. Once initialized, x is assigned to the "NULL" location of head->next and the while-loop breaks, or terminates, in the if-statement.

A momentary reflection yields the observation that head and head->next are completely allocated by malloc. Still, in initchararrll there is a lack of assignment; dynamic memory is allocated because of malloc's call to the operating system, not the assignment of electronic information from standard input to computer memory. It is possible to see in the next section, however, how assignment occurs in HIP.

15. Parsing Standard Input to Linked-List Data Members

Of all the HIP sections sequentially constructed thus far in the thesis, this current section is the most critical for the sustained use of HIP in practical applications. The "Echo Allocation Pointer" echoAllocPtr function is denoted in Appendix G and error checks before iteratively creating structs and assigning standard input, stdin, to them. The struct chararr * hd assigned to the head of the inputted linked list is returned at the end of echoAllocPtr as a method by which the user can access the entirety of the linked list's contents. The argument of the while-loop assigns the contents of head -> intContent one character from the file pointer in question. While a file pointer can be practically anything including a .txt or .dat file, standard input, stdin, is a file pointer. The call to getc, then, picks one character from the stdin file pointer, assigns it to head's intContent, and finally compares head's intContent to the newline sentential '\n' like fgets does for fixed-length arrays contiguous in memory-space. Unlike fgets, though, there is no integer limiter for the while-loop; the while loop terminates either if head is NULL or if the newline '\n' is reached. This condition on head is important, as iterating through the while-loop will lead to heap depletion. The while-loop, then, terminates when head cannot be allocated "on the fly."

The printf statement outputs the head's "intContent" integer to standard output or the computer monitor in most cases. As previously noted, characters in C are low-level integers that can be parsed by a suitable program. Furthermore, this HIP application somewhat simply "echoes" or outputs the message which was inputted to standard input—the keyboard in most cases.

16. Deleting or Freeing a HIP Linked List

For HIP's intentions and purposes, heap memory tailors a linked list to the input capacities of the computer. Either command-line arguments or keyboard user input are fair game for the functions developed in HIP. When the linked list is filled, the developer can decide whether the character data members can fit into an array for faster processing. Beyond this, developers may want to rid themselves of the linked list to "free up" vital system resources. The terse function "Free Linked List" freell iteratively calls the C standard library function free as an inverse of malloc to release system resources dynamically allocated by malloc. The freell function is not recursive, as it was in the alpha testing version of HIP, but rather iterative. The difference is speed; iterative functions are mostly faster than recursive ones and are necessary for parallel computing in such frameworks as the ubiquitous Message Passing Interface MPI.

The freell function involves error checking. Because it is not advisable in C to free a pointer either not allocated by malloc or a NULL pointer, error checking for free must also be implemented according to the conservative prescriptions in (Daconta, 1993, p. 69); if the pointer is not NULL, Daconta's logic prescribes the freeing of the dynamically-allocated pointer. It is up to the developer not to present a statically-allocated pointer (e.g. "int x = 9; int * y = &x;" where & yields the address of x) to free, as this would create many problems for end-user.

With these concepts in mind, presenting an example of the error checking mechanism and an abbreviated form of the freell will significantly contribute to the discussion. The code in freell is in line with Daconta's prescription of enclosing a free function with an if-statement. The else statement is a HIP innovation; if the place pointer

is not a pointer allocated to something—if the pointer is in fact NULL—then the program aborts, as something very erroneous has happened.

While the issue of linked-list deletion presented in (Daconta, 1993, pp. 136, 151) using a for-loop is well-suited for this deletion problem, several prescriptions in (Kernighan & Pike, 1999, p. 48) note that the elements in each linked list node must be deleted or freed "elsewhere." The author (NWK) of this report understands the Kernighan-Pike prescription to mean that the freeing ought not to occur in the same loop containing the linked-list deletion of nodes. HIP follows this convention in Appendix G in deleting the linked list.

This HIP procedure involves a denotation of a temporary node "tmp" designed to be a memory placeholder. Sending dynamically-allocated "tmp" memory to the heap via free is facilitated through iterating tmp along with the head structure pointer. The lines involving "tmp->next = head->next" advance the temporary linked-list node to the new node to which head is pointing. Calling free for each member of tmp and tmp itself is necessary for the proper functioning of any program using HIP; without the proper deallocation of the singly linked list structure, there would be a memory leak within the HIP program leading to problems (e.g. virtual memory issues, computer crashes, etc.).

17. Future Work in Buffer Overflow Research

Closing this BO section is a need to address what future computer-scientific developments might progress from the ideas contained in this thesis. The safe strcat and safe strcpy functions fleshed out in this thesis are expandable to variable-argument functions in C using the stdarg.h header. Programming a variable argument strcat or strcpy without consideration of the ideas in this thesis is haphazard at best. Any attempt to deal with strcat would have to address a prior work (Holub, 1990, p. 746) available also online.

Attempts to address an stdarg.h strcpy would have to proceed from scratch, so far as the author of this thesis (NWK) can ascertain.

Chapter 3: SQL Injection for CS I and CS II Courses

1. Introduction

The prior discussion touching on buffer overflows (BO) would be incomplete if the communication between the database and the server-side scripting or compiled server-side language were left untouched. SQL injection (SQLI) is a phenomenon in which an executed server-side command results in the execution of an undesired SQL statement (e.g. an unwanted table or even database drop). SQLI used to sniff, insert, or delete records into databases has undesirable consequences for developers and users.

One basic SQLI attack which can be covered in CS I and CS II classes is the '1=1' attack which proceeds to emend an SQL clause by adding an " OR '1=1' " conditional at the end of a "WHERE" clause. Proper execution of the WHERE clause, for example "WHERE username='admin' ", does not matter for the '1=1' attack, as '1=1' will always execute as true for every entry in the database.

While SQLI can happen in any programming language, an offline web module demo suitable for CS I and CS II classes involving SQLI may best be presented by using the PHP scripting language and the open-source version of MySQL. PHP in this context has the advantage of being less verbose than the Model-View-Controller forms of C# ASP.NET and Java Spring frameworks. CS I and CS II students may directly observe the database queries and their effects on the development of this demo.

The application presented in this thesis implements an offline C++ program accessing an SQLite database. Students observing this database for the first time shall observe its simplicity while those familiar with databases will note that even complicated databases and simple ones are prone to cyberattack in an application if the server-client interface is designed in deficit. Each offline web module is designed to be loosely coupled so that instructors may choose only the content which adheres to their course plans.

2. SQLite Databases, C++, and SQL Injection

In the previous BO sections, the development of programs useful for C and C++ programmers included many low-level prescriptions perhaps efficacious for command-line application development. Many web applications developed today utilize server-side and client-side languages built on C and C++ such as JavaScript, with many large-scale applications using databases for record and bookkeeping purposes. It is this last point that is the present concern of this thesis. The general introduction to this work discussed the Equifax cyberattack as a vehicle of raising concerns about web development and web activity. The testing of databases is of the utmost concern to developers; consequently, cybersecurity instructors intent on providing an excellent, yet time effective, demonstration to students about the dangers of improperly constructed database record retrieval have an extra pedagogical method in their arsenal.

SQLite is a database well-suited for the classroom demonstration of SQL injection (SQLI or SQLi). It is lightweight, portable, and notable for its easy use. Unlike Postgre SQL, which supports the entire SQL standard, SQLite, like MySQL and many other databases, supports a substantial portion of the SQL standard. Additionally, many programming languages access SQLite either in the development of a full-fledged web

application or in the development of a command-line program that accesses the SQLite database. Nevertheless, it is the flexibility of SQLite which makes it the choice for this venture into the demonstration of SQLI.

It is relevant to note that the use of C++ with SQLite to build command-line programs is ideal for the instructor in the classroom. Unlike C, which is a small, lower-level language with fewer tools, C++, not as low-level as C, is an extensive language with many tools for the instructor. Because of the vastness of C++, the development of the SQLI instructional program will not undergo extensive discussion. Instead, the complexity of a complete program necessitates that the development of C++ machinery will be relegated to references. Templates (Prata, 2002); member function pointers (Deitel & Deitel, 2001, pp. 1090-1092) (Terribile, 1994, pp. 108-111) (Ellis & Stroustrup, 1990, pp. 70-71) (Swan, 1993, pp. 563-566); pairs; and lambda expressions all bring out the inner workings of a full SQLite instructional SQLI database program.

The SQLite wrapper for C++, fortunately, allows the developer to use the flexibility of C++ to develop full-scale database applications. C++ compilers, like C compilers, are all very similar. The portability of SQLite applications developed in C++ extends for this thesis to the GNU Compiler Chain (GCC) of C, Java, Perl, and C++ compilers. Indeed, the testing of this SQLite database program was performed with the Ubuntu Linux 18.04 and 20.04 operating systems using the Atom text editor and the GCC 7.5.0. Portability, especially using the member-function pointers, between compilers is impacted when other C++ compilers are used to compile the program, notably Visual C++. While the GNU C Compiler gcc and the GNU C++ Compiler g++ support SQLite development through wrappers, Visual C++ 2019 is at the time of this writing is a recalcitrant exception.

3. SQL Injection Attacks: Principles and Examples

One basic SQLI attack which can be covered in CS I and CS II classes is the '1=1' attack (Stuttard & Pinto, 2008, pp. 242-243) which proceeds to emend an SQL clause by adding an "OR '1=1' " conditional at the end of a "WHERE" clause. Proper execution of the WHERE clause, for example "WHERE username='admin'", does not matter for the '1=1' attack, as '1=1' will always execute as true for every entry in the database. The '1=1' attack generalizes as well to 'a' = 'a' and so on.

Improperly structured login accounts can be susceptible to the 1=1 attack. An example from (Stuttard & Pinto, 2008, p. 243) is particularly revealing. If the attacker knows the login token of the administrator but not the administrator's password, for instance, an innocuous SELECT query, which returns some or all entries in a database, may turn out to reveal more information than the administrator would have wished. Suppose the original query (Stuttard & Pinto, 2008, p. 243) is a SELECT query that locates a username of 'admin' and a password of 'clandestine.' Such a query necessarily logs the admin into the system. Furthermore, a SELECT * query picks out every entry in a database table 'Table.' Under this assumption, if the attacker knows the admin's login, or any other user for that matter, the attacker may garner copious amounts of information from the SQLite database by using the comment operator. Commenting out the password areas of an ill-constructed SQL query is dangerous if there is a web interface. In this case, the attacker simply enters 'admin' /* into the username textbox and, if fortunate, observes either client-side scripts or server-side programs outputting massive amounts of information to the console. The only thing that matters to the attacker is that the SELECT

statement returned the login credentials to him. INSERT, UPDATE, DELETE, and UNION are all susceptible to SQLI attacks (Stuttard & Pinto, 2008, pp. 247-251).

Another attack that can occur with some databases but not all is a multi-line or semicolon attack (Steinmetz & Ward, 2008, pp. 35-36). The idea behind the semicolon attack is the creation of multiple SQL statements in an input area that is designed to ideally admit one statement. The attacker, building more queries into his target, may decide to employ DROP TABLE or DROP DATABASE in his or her electronic rampage. The outcome for the attacker, if drops are used, is pure destruction. The development of this attack and other attacks on SQLite is treated in a real-life situation requires planning and skill, perhaps even an accomplice who knows some inside information.

4. SQL Query Attacks in SQLI Database SQLite Instructional Program (SIDSIP)

5. Introduction

The SQLI Database SQLite Instructional Program (SIDSIP) implements SQLite SQL to illustrate to interested end viewers the dynamics of an SQLI simulation. In the five forgoing simulations, the table DEMO is defined as a table containing an ID, the NAME of a network user, and his EMAIL for authentication purposes. The STORE table contains an ID, the name of an ITEM, and the QUANTITY of the on-hand product. The SQLite commands implemented for SQLI demos are divided into five SQL queries as noted in Appendix H.

Each SQLI query in Appendix H sketches out security vulnerabilities in SQLite. DELETE FROM illustrates the basics of a 1 = 1 attack. The 1=1 and ‘A’ = ‘A’ attacks are respectively single-line and semicolon attacks. A security flaw in a login protocol is

illustrated in the 'jim' attack. The last two attacks are a 1 = 1 attack with the target being an integer carrier. The final attack illustrates the vulnerabilities of SQLite and the necessity of an SQLite user to properly sanitize inputs into the database.

SIDSIP includes several C++ files—demo.h, queryClasses.h, queryTemplates.h, io.h, and a main.cpp file. For thesis length restrictions, an abbreviated SIDSIP program covering the '1=1' attack is presented in Appendix I.

6. Summary of SIDSIP Functions and Classes

The SIDSIP program contains many functions in need of a brief description. Working from the start of Appendix I, the global function callback is a C-based function that uses function pointers to retrieve entries from an SQLite3 database. While a thorough discussion of function pointers is beyond the scope of this thesis, it is beneficial to observe that the function callback is necessary for the proper functioning of the program. Consequently, some SQLite3 database queries will use the callback function while others will not. There is a fundamental division in the earliest stages of software development between functions in need of callback and functions that do not need a callback. This idea will undergo further development. The remaining global function wait is used to pause the input stream so that the instructor may lecture students about SQL injection at his or her leisure.

7. Console Sanitized Class

The interface class consoleSan is an abbreviation for consoleSanitized, a name denoting the need for the vetting of the input after the user enters something into standard input—the keyboard in most cases. The function tolowercase uses C++ lambda expressions

to change all the characters in user input to their lowercase counterparts: Y to y or No to no. The function yorn stands for “yes or no” and checks the contents of user entries into the standard input. If the user, the instructor, enters “yes,” “y,” “no,” or “n” into standard input, yorn returns to the program either “y” or “n.” The yorn function repeats an answer-reformatting request so long as the user fails to enter an acceptable C-String into standard input. The C macros at the beginning of the program in the preprocessor directives control how many iterations the do-while loop executes before yorn terminates.

8. Database Actions Class

The class dbActions is an abbreviation for “Database Actions” and includes functions that create the database and a simple table. Additionally, auxiliary functions designed to show all the records in a demo database table and insert certain preset values into a demo database table. Finally, the one equals one attack function is the primary function container that executes the queries for the instructor. To make the program functions as simple as possible, the only variable parameters called by functions in dbActions are the SQLite3 database pointer and the errorHandler integer pointer. The inclusion of the database pointer is necessary for the updating and execution of the SQLite3 queries. In fact, not returning an updated SQLite3 db pointer to the program upon the completion of a function’s execution prevents the program from using the updated SQLite3 database. The errorHandler is necessary for the successful error checking of each query performed in the program.

The auxiliary functions each contain error checking and SQLite3 queries. The errorHandler integer pointer registers an error code regardless of the execution state of the query. If errorHandler fails the error code test with SQLITE_OK, then the program displays

an error message and proceeds to free the error message with sqlite3_free. In some implementations of SQLite3 programs, the program will abort abnormally using exit(1). The oneEqualsOne attack function is an amalgamation of the auxiliary functions used in the program and a delete SQLite3 query used to illustrate 1 = 1.

A brief analysis of the class defaultValueDemoDb will illustrate the how the SQLite3 database is initialized and created in SIDSIP.

9. Default Value Demo Database Class

The defaultValueDemoDb class contains a constructor that creates the SQLite3 database used for the demo along with a table populated with default values. The destructor for defaultValueDemoDb closes the SQLite3 database when the instructor needs a fresh database to run a new demo. There are getters in the Default Value Demo Database as well; getDefaultValueDb() and getErrorHandler() both respectively return to the program the SQLite3 database and the error handler integer.

10. Main Function of SIDSIP

Many commercial or open-source C or C++ programs contain very few instructions in the main C++ function: the opening point for the C++ program. SIDSIP is an obvious exception due to its lack of complexity. While the main function does not support command-line arguments, nevertheless, integer argument count “argc” and character pointer array “argv,” or argument vector, are included. A future task would involve expanding the use of argc and argv for the selection of different demo programs.

The wait() function is used here to give the instructor the ability to pause the demo. The C++ strings “choice double table” choiceDbTbl, choiceOne, choices, and

choiceDbTblch or “choice double table character” are all used in the main function to record user choices. In a more complicated version of SIDSIP, a do-while loop is used to give the instructor the option of presenting more than one demo to a class. In this simplified program, however, one if statement is used to check whether “y” or “n” is entered by the user.

The line in which choices are assigned is critical to the execution of the program. The function yorn in the consoleSan function was declared to be static; yorn can be called without explicitly having to use a C++ object. The arguments in yorn assign a value to choiceDbTbl within the yorn function and provide an upper and lower bound to the number of times the instructor may incorrectly enter a response into standard input. The string choices is then assigned the returned contents of consoleSan::yorn. Comparison between choices and “y” is essential for the functioning of the program. There would be no input sanitization in the program without calling consoleSan::yorn.

Once in the if statement, a new default value demo database pointer “db” is created and the contents of choiceDbTblch are reset. A second choice is presented to the instructor as to whether the one-equals-one attack should proceed. If the answer is “y,” the queries are constructed for the one-equals-one attack from SELECT and DELETE statements. Finally, the “db” object inheriting the dbActions class shows all the records prior to the attack and all the records after the attack. In the full SIDSIP program, the “db” object is deleted and recycled in a do-while loop. The return 0 customarily ends the program.

11. SQL Injection Conclusions

SIDSIP is unsettling. The 5 attacks are successful to unsuspecting victims if their queries are ill-constructed. A more complicated program designed for C++ developers familiar with SQLite3 using advanced features such as member function pointers and templates is available from the author of this thesis (NWK). Online resources to SQLite3 can be found online at websites such as tutorialspoint (SQLite Tutorial - tutorialspoint, 2021) and the main SQLite website (SQLite Home Page, 2021).

Future work in this area includes creating a graphical user interface for the program. At the time of this writing, Qt is a natural C++ choice for cross-platform applications.

REFERENCES

Daconta, M. C. (1993). *C Pointers and Dynamic Memory Management.* Boston: QED Publishing Group.

Deitel, H. M., & Deitel, P. J. (2001). *C++: How to Program* (3rd ed.). Upper Saddle River: Prentice Hall.

Du, W. (2017). *Computer Security.* Lexington: CreateSpace.

Ellis, M. A., & Stroustrup, B. (1990). *The Annotated C++ Reference Manual.* Reading: Addison-Wesley.

Forouzan, B. A., & Gilberg, R. F. (2001). *Programming Approach Using C* (2nd ed.). Pacific Grove: Brooks/Cole Thomson Learning.

Harbison, S. P., & Steele, G. L. (1991). *C: A Reference Manual* (3rd ed.). Englewood Cliffs: Prentice-Hall.

Holub, A. I. (1990). *Complier Design in C.* Englewood Cliffs: Prentice-Hall.

Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language* (2nd ed.). Englewood Cliffs: Prentice Hall PTR.

Kernighan, B. W., & Pike, R. (1999). *The Practice of Programming.* Reading: Addison-Wesley.

Kernighan, B. W., & Ritchie, D. M. (1978). *The C Programming Language.* Englewood Cliffs, New Jeresy: Prentice Hall.

Kruse, R. L., Leung, B. P., & Tondo, C. L. (1991). *Data Structures and Program Design in C.* Englewood Cliffs: Prentice-Hall.

Lafore, R. (1987). *Microsoft C: Programming for the IBM.* Indianapolis: Howard W. Sams & Company.

Miller, C., & Zovi, D. D. (2009). *The Mac Hacker's Handbook.* Indianapolis: Wiley.

Pfleeger, C. P., & Pfleeger, S. L. (2007). *Security in Computing* (4th ed.). Westford: Prentice-Hall.

Prata, S. (2002). *C++ Primer Plus.* Indianapolis: Sams.

Shahab, A., Alenezi, M., Nadeem, M., & Asif, R. (2020). An automated approach to fix buffer overflows. *International Journal of Electrical and Computer Engineering*, 3777-3787.

Shaw, A. (2014). Program Transformations to Fix C Buffer Overflow. *ICSE Companion 2014: Companion Proceedings of the 36th International Conference on Software Engineering*, 733-735.

SQLite Home Page. (2021, January 20). Retrieved March 10, 2021, from https://www.sqlite.org/index.html

SQLite Tutorial - tutorialspoint. (2021). Retrieved March 10, 2021, from tutorialspoint: https://www.tutorialspoint.com/sqlite/index.htm

Steinmetz, W., & Ward, B. (2008). *Wicked Cool PHP.* San Francisco: No Starch Press.

Stuttard, D., & Pinto, M. (2008). *The Web Application Hacker's Handbook.* Indianapolis: Wiley.

Stuttard, D., & Pinto, M. (2011). *The Web Application Hacker's Handbook* (2nd ed.). Indianapolis: John Wiley and Sons.

Swan, T. (1993). *Learning C++.* Carmel: 1993.

Terribile, M. A. (1994). *Practical C++.* New York: McGraw-Hill.

Tjaden, B. C. (2004). *Fundamentals of Secure Computer Systems.* Wilsonville, Oregon: Franklin, Beedle & Associates.

Waite, M., & Prata, S. (1990). *C: Step-by-Step.* Carmel: Howard W. Sams & Company.

Waite, M., Prata, S., & Martin, D. (1989). *C Primer Plus* (Revised ed.). Indianapolis: Howard W. Sams & Company.

Wikman, E. C. (2020). *Static Analysis Tools for Detecting Stack-Based Buffer Overflows.* Monterey: Naval Postgraduate School.

Zhao, Y., Du, X., Krishnan, P., & Cristina, C. (2018). Buffer Overflow Detection for C Programs is Hard to Learn. *Proceedingsvof (ISSTA Companion/ECOOP Companion' 18*, 2.

APPENDIX A

Examples of a Buffer Overflow with String Concatenation “strcat”

```
#include "stdio.h"
#include "string.h"

void strcatSandwich()
{
	char type[14]; //does not work for "chicken"
	char sandwich[] = " sandwich";
	printf("Enter the name of the sandwich you want:\n");
	scanf("%s", type);
	printf("Here is your receipt:\n");
	printf("%s", strcat(type, sandwich));
}

int main ()
{
	char* bose = "Bose";
	char* chandrasekhar = "Chandrasekhar";
	char greetUser[15] = "Greetings ";
	char greetUserTwo[15] = "Greetings ";
	printf("%s",strcat(greetUser,bose));
	printf("%s", strcat(greetUserTwo, chandrasekhar));
	return 0;
}
```

APPENDIX B

Program “Safe String Concatenation” sfstrcat

```
void sfstrcat(char s[], int lens, char t[], int lent)
{
	/*Notice: If lens is greater than the actual size of the char array, then we will have a buffer overflow!*/
	int i, j;
	i = j = 0;

	if ((lens - strlen(s) - 2) >= lent) //Normal strcat operations. Notice the null termintors terms on the LHS...
	{  /*From K&R*/
		while (s[i] != '\0')
			i++;
		while (s[i++] = t[j++])
			;
	}
	else if ((lens - strlen(s) - 2) < lent) //If lent is greater than the extra space specified in lens, then this code avoids a buffer overflow...
	{
		while (s[i] != '\0')
			i++;
		int y = lens - strlen(s) - 2;
		while (y-- >= 0)
		{
			s[i++] = t[j++];
			if (y == 0)
			{
				s[i] = '\0';
				break;
			}
		}
	}
}
```

APPENDIX C

Program “Modified Safe String Concatenation” modsfstrcat

```
char* modsfstrcat(char s[], int lens, char t[], int lent)
{
        int i = 0, j = 0, maxt, spaces = lens - strlen(s);
        if (spaces-- > 0 && (maxt = lent <= strlen(t) + 1 ? lent : strlen(t) + 1) > 0) //Notice
the null-terminator next to strlen(t)
                //Ideally, the inputs lens and lent are the maximum number of bytes
respectively contained in char s[] and char t[]
        {
                while (s[i])
                        i++;
                while (--spaces >= 0 && --maxt >= 0 && (s[i++] = t[j++]))
                        ;
        }
        return s;
}
```

APPENDIX D

Program “Pointer Safe String Concatenation”

```
char* ptrstrcat(char* s, int lens, char* t, int lent)
{
        int i = 0, j = 0, maxt, spaces = lens - strlen(s);
        char *sstart = s, *tstart = t;
        if (spaces-- > 0 && (maxt = lent <= strlen(t) + 1 ? lent : strlen(t) + 1) > 0)
        {
                while (*s)
                        s++;
                char* sbeginCat = s;
                while (s - sbeginCat < spaces && (*s++ = *t++) && t - tstart < maxt)
                        ;
        }
        return sstart;
}
```

APPENDIX E

Program “String Copy Matrix:” An Example of Buffer Overflow with String Copy

```
void strcpyMatrix()
{
	char exit;
	int qty = 10;
	char original[] = "1 1\n2 2"; // 8 characters long
	char final[9]; //must be 9 or greater!
	strcpy(final, original); //copy original matrix; buffer overflow.

	for (int i = 0; original[i] != '\0'; ++i)
	{
		printf("%c", final[i]);
		printf("\n");
	}
}
```

APPENDIX F

Program “Safe String N-Copy”

```
char* sfstrncpy(char* d, int lend, char* s, int lens)
{
        int choices = lens <= strlen(s) + 1 ? lens : strlen(s) + 1;
        char* dbegin = d;
        if (lend <= choices)
        {
                while (lend-- > 1 && (*d++ = *s++))
                        ;
                *d = '\0';
        }
        else
        {
                while (*d++ = *s++)
                        ;
        }
        return dbegin;
}
```

APPENDIX G

Program HIP (Heap Input Parsing) Program

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

struct chararr
{
	int* place;
	char* chrContent;
	int* intContent;
	struct chararr* next;
};

struct chararr* chkAllocChararr(struct chararr* ca) //Check Allocation of Char Array
{
	if (!ca)
	{
		fprintf(stderr, "Fatal Error! Dynamic Memory Allocation Failed!");
		exit(1);
	}
	return ca;
}

int* chkAllocInt(int* i) //Check Allocation of Integer
{
	if (!i)
	{
		fprintf(stderr, "Fatal Error! Dynamic Memory Allocation Failed!");
		exit(1);
	}
	return i;
}

char* chkAllocChar(char* ch) //Check Allocation of Character
{
	if (!ch)
	{
		fprintf(stderr, "Fatal Error! Dynamic Memory Allocation Failed!");
		exit(1);
	}
	return ch;
}
```

```
struct charrarr* initcharararrll() //Initialization of Linked-List Character Array
{
	struct chararr* x = (struct chararr*)malloc(sizeof(struct chararr)); //Allocation of
struct
	chkAllocChararr(x); //Error Checking

	(x->intContent) = (int*)malloc(sizeof(int)); //int allocation for integer content
	chkAllocInt(x->intContent);

	(x->place) = (int*)malloc(sizeof(int)); //int allocation for placeholder
	chkAllocInt(x->place);

	x->chrContent = (char*)malloc(sizeof(char)); //char content allocation
	chkAllocChar(x->chrContent);

	x->next = NULL; //the next node doesn't point to anything!

	return x;
}

struct chararr* addNodeReturnLastNonNullNode(struct chararr* head)
{
	if (!head) //error checking for head of linked list
	{
		fprintf(stderr, "NULL pointer inserted in linked list! Exiting...\n");
		exit(1);
	}
	else
		while (head)    //while head is not null…
		{
			if (!head->next)  //if next is null…
			{
				struct chararr* x = initcharararrll();//(&place);
				head->next = x;
				break;
			}
			head = head->next;
		}
	return head; // We're at the end of everything which is initialized in the linked-list!
}

struct chararr* echoAllocPtr(struct chararr* head, int initPlace, FILE* stream)
{
	if (head == NULL) //if head is null…
	{
		printf("Fatal Error: Null or Uninitialized Pointer Passed to Function!");
```

```
                exit(1); //abnormal termination of program
        }
        else
        {
                struct chararr* hd = head;
                while ((*(head->intContent) = getc(stream)) != '\n' && head) //Gather
input from either standard input or another input stream.
                {
                        printf("%c", *(head->intContent)); //Display
                        *(head->chrContent) = *(head->intContent); //Assign the low-level
integer input to the chrContent character.
                        *(head->place) = initPlace++; //Assign to place initPlace and
Increment initPlace on the next line.
                        head = addNodeReturnLastNonNullNode(head); //Add a new node
at the end of the linked list.
                        head = head->next; //Assign head to next linked list member.
                }
                return hd;
        }
}

void freell(struct chararr* head)
{
        if (!head) //Error checking for head
        {
                fprintf(stderr, "Null pointer passed to be freed! Exiting...\n");
                exit(1); //abnormal termination
        }
        else
        {
                for (struct chararr* tmp = head; tmp != NULL; tmp = tmp->next) //free
individual elements from linked list
                {
                        if (tmp->place) //Following Daconta's error checking prescriptions
                                free(tmp->place);
                        else
                        {
                                fprintf(stderr, "Fatal error! Aborting...\n");
                                exit(1);
                        }
                        if (tmp->chrContent)
                                free(tmp->chrContent);
                        else
                        {
                                fprintf(stderr, "Fatal error! Aborting...\n");
                                exit(1);
```

```
                }
                if (tmp->intContent)
                        free(tmp->intContent);
                else
                {
                        fprintf(stderr, "Fatal error! Aborting...\n");
                        exit(1);
                }
        }

        struct charrarr* next; //Kernighan, Ritchie, and Pike
        for (; head != NULL; head = next) //free linked list nodes
        {
                next = head->next;
                if (head)
                        free(head);
                else
                {
                        fprintf(stderr, "Fatal error! Aborting...\n");
                        exit(1);
                }
        }
    }
}

int main(int argc, char** argv) //program execution begins here. Prints all chars in linked
list
{
    printf("Enter a string to echo it to the console:\n");
    int iniPlace = 0;
    struct chararr* head = initchararrll(&iniPlace);
    struct chararr* hd = head;
    head = echoAllocPtr(head, iniPlace, stdin);

    printf("\nNow let's run through the string to show the positions of the individual
characters...\n");
    for (struct chararr* i = head; i->next != NULL; i = i->next)
        printf("char %c is at place %d\n", *(i->chrContent), *(i->place));
    freell(hd);
    return 0;
}
```

APPENDIX H

SQLite3 Queries Used in Full Version of SQLI Database SQLite Instructional Program (SIDSIP)

```
DELETE FROM table WHERE EMAIL = 1 = 1;/*email@website.com';

SELECT * FROM DEMO WHERE NAME = 'A' = 'A'; DROP TABLE DEMO; /*';

SELECT * FROM table WHERE username = 'jim'; /*' and password = 'something';

SELECT * FROM STORE WHERE QUANTITY >= 1 = 1; /* AND ITEM = 1;
```

APPENDIX I

G++/Linux Source Code For An Abbreviated SIDSIP Program

```
#include <iostream> //Input Output for C++
#include <string> //Useful functions for evaluating C++ strings
#include <sqlite3.h> //Necessary for using SQLite3 databases
#include <algorithm> //contains for_each. useful for manipulating C++ strings
#include <cstdio>   //C-style standard input and output
#define minMenu 1     //C Macro used for giving the minimum number of main function do-while loop to go through
#define maxMenu 30  //C Macro used for giving the maximum number of main function do-while loop to go through

using namespace std;
//https://www.tutorialspoint.com/sqlite/sqlite_c_cpp.htm
//Compile with...
//g++ program.cpp -lsqlite3 -o program.exe

//callback is used in C or C++ SQLite3 wrappers. Some SQLite3 functions have callbacks. Others do not.
static int callback(void* NotUsed, int argc, char** argv, char** azColName) {
   int i;
   for (i = 0; i < argc; i++) {
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
      //in the printf statements are function pointers that interface with <sqlite3>
   }
   printf("\n");
   return 0;
}

//Pauses the input and awaits the instructor to enter a character or new line.
void wait()
{
   int input;
   while ((input = getchar()) != '\n') //An old trick in C from K&R.
      ;
}

//consoleSanitized:
class consoleSan {
public:
   string static tolowercase(string hw)
      //tolowercase takes a C++ string and sets all of its characters to lowercase.
   {
      string hwlwr = ""; //an empty string
```

```
        for_each(hw.begin(), hw.end(), [&hwlwr](char c) {hwlwr.push_back(tolower(c));
});
        //Notice the lambda expression in the line above. Each C character comprising the
C++ string is converted to lowercase.
        return hwlwr;//hello world lwr is returned as opposed to hw
    }

    string static yorn(string choice, int exitctr, int numDisplay) //yes or no.
    {
        cout << "Y|n: "; //outputs to the screen the prompt for yes or no.
        string yes = "yes"; //to be compared with choice
        string y = "y";//to be compared with choice
        string n = "n";//to be compared with choice
        string no = "no";//to be compared with choice
        cin >> choice; //Notice that choice is external to the function. cin >> here is
simplest.
        cout << endl;
        string lwrcase = tolowercase(choice); //tolowercase is the first step of sanitization
        do
        {
            if (lwrcase.compare(n) == 0) //if lwrcase matches with 'n' then break out of the
loop.
                break;
            else if (lwrcase.compare(y) == 0) //if lwrcase matches with 'y' then break out of
the loop.
                break;
            else //a poor man's error checking else clause
            {
                cout << "Please reformat your answer. Y|n:" << endl;
                cin >> choice; //we now intake a new user input into the program.
                lwrcase = tolowercase(choice); //lwrcase is recycled and the program continues.
                cout << endl;
            }
        } while (--exitctr >= numDisplay); //exitctr is decremented in the statement
argument and is checked with numDisplay
        //the number of Display integer is the limiting factor
        if (lwrcase == "no" || lwrcase == "n")
            return n;
        else
            return y;

    }
};

//class Database Actions contains functions which create an SQLite3 database, create a
blank table,
```

```
//insert into the table preset values, show all db records, and a demo of the 1 = 1 attack.
class dbActions {
public:
   sqlite3* createDbAndCheckForErrors(sqlite3* db, int* errorHandler, string dbName)
//create a db and check for errors
   {
      cout << "Enter a name for the database or enter .db for the dbName testsqli.db" <<
endl;
      cin >> dbName; //Notice in this function that dbName is not a function variable
parameter
      //the errorHandler below uses the ternary conditional operator to open/create a
default sqlite3 db.
      //errorHandler in the function call is a pointer, actually.
      (*errorHandler) = (dbName.c_str() == ".db") ? sqlite3_open_v2("testsqli.db", &db,
SQLITE_OPEN_CREATE | SQLITE_OPEN_READWRITE, NULL) :
sqlite3_open_v2(dbName.c_str(), &db, SQLITE_OPEN_CREATE |
SQLITE_OPEN_READWRITE, NULL);
      if (*errorHandler != SQLITE_OK) { //if something goes wrong at this stage, abort
the program immediately!
         fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
         sqlite3_close(db);//db should be recycled here and freed.
         exit(1);
      }
      else {
         fprintf(stdout, "Opened database successfully.\n");
      }
      return db; //necessary for updating and maintaining the state of the database
   }

   //creates an empty sqlite3 table and checks for errors
   sqlite3* createSimpleTblAndChkForErrors(sqlite3* db, int* retTable)
   {
      char* errorMsg = NULL; //this C-String is always filled with a success or error
message by createSimpleTbl
      cout << "Creating a Demo Table Named: " << endl;
      cout << "DEMO...\n" << endl;
      //Now for the sqlite3 query:
      string demoName = "CREATE TABLE DEMO(ID INT PRIMARY KEY NOT
NULL, NAME TEXT NOT NULL, EMAIL CHAR(50) NOT NULL);"; //SQL Query
C++ string
      cout << demoName << endl; //output the sqlite3 query string
      (*retTable) = sqlite3_exec(db, demoName.c_str(), NULL, 0, &errorMsg); //retTable
is dereferenced and assigned to the return value of sqlite3_exec()
      cout << endl;
      if (*retTable != SQLITE_OK) //C-Based error checking
      {
```

```
            fprintf(stderr, "SQL error: %s\n", errorMsg);
            sqlite3_free(errorMsg); //the errorMsg pointer must be freed.
        }
        else //Normal operation
        {
            fprintf(stdout, "Table created successfully\n");
        }
        return db; //updating db
    }

    //populates an SQLite3 table with a series of preset values
    sqlite3* insertIntoDefaultTableValues(sqlite3* db, int* errorHandler)
    {
        char* errorMsg = NULL; //success or errorMsg
        string sql = "INSERT OR REPLACE INTO DEMO(ID, NAME, EMAIL) VALUES
(1, 'John', 'john@website.com');"
            "INSERT OR REPLACE INTO DEMO(ID, NAME, EMAIL) VALUES (2, 'Jeff',
'jeff@website.com');"
            "INSERT OR REPLACE INTO DEMO(ID, NAME, EMAIL) VALUES (3, 'Joe',
'joe@website.com');"
            "INSERT OR REPLACE INTO DEMO(ID, NAME, EMAIL) VALUES (4, 'Jim',
'jim@website.com');";
        //this long-winded multi-lined SQLite3 query populates the database table
        cout << "The SQL statements are:" << endl;
        cout << sql << endl; //one little command displays the entire C++ string
        cout << endl;

        (*errorHandler) = sqlite3_exec(db, sql.c_str(), NULL, 0, &errorMsg);
        //errorHandler is assigned to sqlite3_exec's execution of the population request
        wait(); //wait() is defined above
        cout << endl;

        if (*errorHandler != SQLITE_OK) //if the population operation fails...
        {
            fprintf(stderr, "SQL error: %s\n", errorMsg);
            sqlite3_free(errorMsg);
        }
        else {//success
            fprintf(stdout, "Records created successfully\n");
            cout << "Records are: " << "(1, 'John', 'john@website.com')" << endl;
            cout << "Records are: " << "(2, 'Jeff', 'jeff@website.com')" << endl;
            cout << "Records are: " << "(3, 'Joe', 'joe@website.com')" << endl;
            cout << "Records are: " << "(4, 'Jim', 'jim@website.com')" << endl;
            wait();
        }
        return db; //updating db
```

```
    }

    //This function reveals all the records in the SQLite3 db
    sqlite3* showAllRecords(sqlite3* db, int* errorHandler)
    {
        char* errorMsg = NULL;
        string sql = "SELECT * FROM DEMO WHERE 1=1;"; //selects all records in db
        cout << sql << endl;
        (*errorHandler) = sqlite3_exec(db, sql.c_str(), callback, (void*)"Callback function
called", &errorMsg);
        //Notice the use of the Callback here. In more lengthy SQLite3 programs, the
developer must know callback
        //and non-callback functions.
        cout << endl;
        if (*errorHandler != SQLITE_OK) {//failure
            fprintf(stderr, "SQL error: %s\n", errorMsg);
            sqlite3_free(errorMsg);
        }
        else { //success

            fprintf(stdout, "All records from database successfully retrieved\n");
        }
        return db;
    }

    //This function executes a 1=1 attack demo.
    sqlite3* oneEqualsOneAttack(sqlite3* db, int* errorHandler)
    {
        char* errorMsg = NULL;
        string chkTableNull = "SELECT COUNT(*) FROM DEMO;"; //SQLite query
tabulates the number of rows in a table.
        cout << endl;
        (*errorHandler) = sqlite3_exec(db, chkTableNull.c_str(), callback, (void*)"Callback
function called", &errorMsg); //the callback for COUNT is necessary here.
        wait();
        cout << endl;
        if (*errorHandler != SQLITE_OK) { //failure
            fprintf(stderr, "SQL error: %s\n", errorMsg);
            sqlite3_free(errorMsg);
        }
        else //success
        {
            fprintf(stdout, "All records from database successfully present\n");
            cout << endl;
            cout << "Now we move on to the 1=1 attack. We'll also see that \'a\' = \'a\' also
does the trick." << endl;
```

```
        string oneEqualsOne = "DELETE FROM DEMO WHERE 1=1;";
        cout << endl;
        cout << oneEqualsOne << endl;
        cout << endl;
        (*errorHandler) = sqlite3_exec(db, oneEqualsOne.c_str(), callback,
(void*)"Callback function called", &errorMsg); //callback necessary for DELETE
        wait();
        if (*errorHandler != SQLITE_OK) //irregularities in the program
        {
           fprintf(stderr, "SQL error: %s\n", errorMsg);
           sqlite3_free(errorMsg);
        }
        else //1=1 successful
        {
           cout << "Let's try to retrieve the records..." << endl;
           showAllRecords(db, errorHandler);
           wait();
           cout << endl;
           cout << "Notice that the SQLite3 database still exists. The records are gone."
<< endl;
           cout << endl;
           cout << "Reviving the records gives us:" << endl; //necessary for a new demo!
           wait();
           db = insertIntoDefaultTableValues(db, errorHandler);
           wait();
           showAllRecords(db, errorHandler);
        }
     }
     return db; //returning db yields a fresh database
  }

};

//this class defaultValue inherits dbActions and is a container for sqlite3* defaultValueDb
class defaultValueDemoDb : public dbActions {
private: //all the data members you need for an sqlite3 * simulation are private here.
   sqlite3* defaultValueDb;
   char* errorMsg = NULL;
   int errorHandler;
   string dbName;
   //These public functions and constructor enable the database to be updated throughout
the program.
   //Notice that the 1=1 attack function is inherited from dbActions
public:
   sqlite3* getDefaultValueDb() { return defaultValueDb; } //getter: returns
defaultValueDb
```

```
   int* getErrorHandler() { return &errorHandler; } //getter: returns errorHandler. Note:
errorHandler is an int not a ptr!
   defaultValueDemoDb() //constructor
   {
      //in each of these calls, defaultValueDb is updated from pointer to pointer.
      defaultValueDb = createDbAndCheckForErrors(defaultValueDb, &errorHandler,
dbName);
      defaultValueDb = createSimpleTblAndChkForErrors(defaultValueDb,
&errorHandler);
      defaultValueDb = insertIntoDefaultTableValues(defaultValueDb, &errorHandler);
   }
   ~defaultValueDemoDb() { //destructor
      sqlite3_close(defaultValueDb); //get rid of the sqlite3 database for the next demo.
   }
};

int main(int argc, char* argv[])
{
   cout << "Hello! Welcome to the SQLite Injection Tutorial!" << endl;
   wait(); //wait asks for the instructor to press the [Enter] carriage
   cout << "We will first create an SQLite Database." << endl;
   cout << endl;
   cout << "Do you want to initialize the database to a set of default values?" << endl;
   cout << "This default-value choice is recommended for an in-class demonstration!" <<
endl;
   string choiceDbTbl = "", choiceOne = "", choices = "", choiceDbTblch = ""; //options
for the program
//choice double table ch assumes the contents of yorn, taking choice double table
   if ((choiceDbTblch = consoleSan::yorn(choiceDbTbl, minMenu, maxMenu)) == "y")
   {
      defaultValueDemoDb* db = new defaultValueDemoDb(); //a database is created on
the heap with C++'s new
      choiceDbTblch = ""; //the contents are reset.
      cout << endl;
      cout << "Do you want to execute a '1' = '1' attack?" << endl;
      if ((choices = consoleSan::yorn(choiceOne, minMenu, maxMenu)) == "y") //
choices assumes yorn's return value
      {
         string oneqry = "SELECT * FROM DEMO WHERE '1'='1';"; //this is the query
that will be displayed
         string oneattk = "DELETE FROM DEMO WHERE '1'='1' # ;"; //this query will
also be displayed
         cout << endl;
         cout << "We will first select all the entries from the database using a SELECT
statement:" << endl;
         wait();
```

```
            db->showAllRecords(db->getDefaultValueDb(), db->getErrorHandler()); //cycles
through and outputs to stdout all the initial records
            wait(); /1=1 attack
            db->oneEqualsOneAttack(db->getDefaultValueDb(), db->getErrorHandler()); /
        }
        delete db; //the database is deleted
    }
    return 0;
}
```