# Buffer Overflow Simulation

Ryan Heathcote

14 December 2012

# 1 Introduction

Most people acquainted with computer security will know that there is such an exploit as the buffer overflow, and that such exploits are often devastating. This exploit can be performed against programs that fail to ensure that buffers are written only within their bounds – that is, they allow data to be written beyond the end of the space allocated for the given variable, and into the space allocated for control information. This vulnerability enables an attacker to take control of the program (and its corresponding permissions), often with devastating effect. The most significant work on the topic is the seminal piece by Aleph One, Smashing the Stack For Fun And Profit.[1] His piece is well worth a read, especially if one does so while attempting to follow his commands on one's own machine. However, due to the complexity of the intel instruction set, following his actions is somewhat difficult. This project aims to simulate the buffer overflow vulnerability in a way that effectively demonstrates the concepts involved in both exploiting and preventing it, while making the exploit simpler to perform than on a full x86-based system.

# 2 How does it work?

The key features of this simulation are: integer-based memory structure, a basic programming language, a compiler, a runtime environment, graphical displays of memory, selectable code and input samples, and toggleable security countermeasures.

## 2.1 Integer-Based Memory Structure

The virtual machine used by the simulation has a main memory consisting of an array of 1000 unsigned integers. Instructions require one integer for opcode, and an additional integer for each parameter (if any). String variables are stored 4 characters to an integer using some bitshifting wizardry. Code is stored at the bottom of the memory structure, starting at cell 0, and grows upwards until the compiler is finished compiling. During runtime, a stack is started at the top of memory (cell 999) and grows downward. The stack consists of a number of stack frames. Each stack frame contains variables and control information for a function call. If any variables are declared in a function, they appear first. A variable has a name cell (an

integer by which it can be identified in read, readn and print instructions), followed by a size sell (in source code size is specified in characters. When compiled, size is in memory cells, so approximately source size divided by four). After the size cell come size number of cells for data for that variable, then either the next variable or a 0 cell. The 0 cell denotes the end of variables for the current stack frame, and also functions as a terminator canary (but is only checked if terminator canaries are selected). If a random or random xor canary is selected, that will come after the 0 cell, followed by the base address of the previous stack frame, the top address of the previous stack frame (used to find the start of the current stack frame) and the return address of the function (the address of the next instruction to be executed after the current function finishes executing).

## 2.2 Language

In order to simulate buffer overflows, a simple language is used that provides minimal, but sufficient, functionality to demonstrate the vulnerability. The language consists of the following instructions:

- dostuff: burns up a virtual cpu cycle and prints "Doing Stuff, Yay!!!" to the virtual screen.
- call <function name>: calls the function identified by function name. Said function must be defined prior to the function call. This call operation involves creating a new stack frame for the new function (using the structure described in the memory structure section). A stack frame is also created when the program begins (this is referred to as the main frame, corresponding to the main function, although this is entirely implicit).
- var <variable name><size>: declares a variable identified by the given variable name, of the size number of characters. all variable declarations must come before any other code in a function. Variables are only accessible within the function in which they are declared.
- function <function name>: indicates the start of a function definition for the function identified by function name. Function definitions need to occur before functions are called, otherwise the compiler will give a -1 address to any calls made before the function definition, as the name will not be registered in the namemap. A call of -1 will likely result in odd occurences. Defining a function twice is also

untested, but will possibly work. It may result in any calls between the definitions pointing to the first function, and any after the second definition pointing to the second function, but this is not certain.

- return: indicates the end of a function definition. A function not having a return would probably cause the compiler to fail. The compiler will not say anything if there are more returns than functions. However, this will result in attempts to return from the main stack frame, which is a security problem if main contains variables, but will normally result in a dirty exit from the program.

- read <variable name>: reads a line (until first

  n) from the input buffer into the variable specified. Will write length of line unless bounds checking is enabled. This is an insecure operation much in the same vein as strcpy() in C.

- readn <variable name><n>: reads n characters from the input buffer and writes them into the specified variable. If bounds checking is enabled, it will write either n or the length of the variable, whichever is smaller. This operation gives the programmer the potential to write secure code, but its security depends on the n parameter being smaller than the size of the target variable, unless bounds checking is enabled.

- print <variable name>: prints the contents of the given variable to the virtual screen until it reaches a null character. This activity may expose memory to some extent, however, the implementation of string storage limits the usefulness hereof, in that an opcode contains three null characters (since the integer value is small enough to affect only the last byte of the memory cell. This implies that a print will stop reading from the variable when it hits an opcode because of the null characters).

- exit: terminates the running of the program. Also prints a message to the screen indicating a clean exit.

- crash: terminates execution and prints a message that the system is crashing to the screen. This instruction is not recognized by the compiler (in other words, the compiler will not compile it into memory), but is recognized by the runtime with the opcode of J (decimal 74). The sole purpose of this command is to give a cool way to prove an exploit. If you can get the runtime to execute a crash instruction, it proves you have circumvented normal operation of the system.

## 2.3 Compiler

The compiler is very simple: it takes source code and translates the instructions and parameters into compiled opcodes and parameter values that the runtime environment can operate on. The compiler puts these values directly into the memory space of the virtual machine. The memory space is not zeroed after each run, so it is entirely possible that this unzeroed memory could be used in an exploit. The following is a list of specific things the compiler does:

- translate function names to addresses: this results in call instructions having the parameter of the address of the first instruction of the function. As the compiler moves through the code, it stores all the names of both functions and variables in a name map, and when a call instruction is encountered, retreives the address of the named function and puts it in as the first operand to the call instruction.

- convert variable sizes: since variables are specified in number of characters in language source, the compiler converts these sizes to the appropriate value for integer cells (divide by four and round up). This conversion makes it much easier for the runtime to operate on variables, since it doesn't have to perform that conversion itself each time.

- translate variable names into integer identifiers: Since variables are allocated in the stack, keeping track of their addresses at compile time is impossible. Thus to manage variables, each variable is given a "name" (an integer identifier starting with 1001), and any runtime variable access will search the current stack frame's variables for that variable name in order to read from or write to a variable. This opens the possibility that an exploit could create its own variables by rewriting the stack frame correctly. One of the input samples demonstrates this by reading to and printing from a variable that was not declared in source within a function that was not defined in source.

- ignore bad instructions: The compiler will give an error message if it encounters an instruction that is not in the defined set of instructions in the language. It will not compile any instructions into memory for that line, but will continue to compile the rest of the source code. This applies also to crash, which, although being defined in the language, is only recognized by the runtime, and not by the compiler.

## 2.4 Runtime environment

The runtime environment starts with memory location 0 and executes code compiled by the compiler. It creates and moves through stack frames as the code directs it. One of the most significant features is the steppable execution mode. While code can be compiled and run in one shot, thus producing instantly the final result of the code and input, it is far more useful from the learning perspective to step through the code line by line, and see the changes in the stack and input buffer that occur after each line of code. This is achieved by separating out the logic for a single iteration of the runtime environment from the loop that repeats these iterations until the code is complete. The "Step" invokes this single step operation once each time it is clicked.

## 2.5 Graphical Memory Displays

Memory is shown in three areas: the code segment, the stack, and the next instruction. The code segment is displayed so that the compiled form of the code can be consulted when performing an exploit. The code segment display is also useful when one wants to write all the way through memory down to the code segment. The stack display shows all the stack frames and the variables and control information for each stack frame. This is perhaps the most useful when performing an exploit. Finally, there is a line that shows what instruction will be executed next, which is useful if one wants to change the input, or for monitoring execution when instructions don't give any screen output.

## 2.6 Code and Input Samples

Both the code edit box and the input buffer edit box have above them a dropdown that enables the user to choose a code sample or input sample to use. These can be useful for quick demonstrations of an exploit, or for figuring out how the system works. Also, below the input buffer is a spin edit that enables putting integer values into the input buffer. Either a full integer can be inserted (which inserts the 4 characters that correspond to that integer, which will properly inject after the string has been written with the bitshifting that occurs), or a single character (with the given ASCII value) can be inserted.

The available code samples are:

### 2.6.1 Code Samples

- secure code: illustrates some basic code written in the language that should not have any buffer overflow vulnerabilities. While it is not intended to be exploitable, it is possible that a vulnerability may exist.
- insecure read: contains a read operation in a function, and is vulnerable to a buffer overflow through said read instruction.
- insecure readn: contains a readn operation that reads more characters than the size of the target variable. The number of characters read in is intentionally large enough to enable overwriting the return address of the current stack frame. While this only allows a small amount of user defined code, this vulnerability could also exploit memory that has already been overwritten in a previous execution of other code that enabled a larger overflow.
- extra return: contains an extra return instruction that does not correspond to a function definition. This vulnerability enables execution of arbitrary code if the user exploits the read operation to overwrite the return address of the main stack frame.
- many functions: has no vulnerabilities because it does not have any read operations. The purpose of this sample is to demonstrate how the stack works with multiple function calls. It could also be used to demonstrate how memory is not initialized, and the consequence thereof that previously written values can be reused for an exploit.

The available input samples are:

### 2.6.2 Input Samples

- benign input: contains short lines that should present no threat to 10 character variables, even with read operations. This would not be benign for really small (less than 8 character) variables.
- code smash: contains enough characters in one line to overwrite memory right down through the code segment. It contains a repetition of the character sequence corresponding to the crash opcode. In normal form, the instruction to be executed next will be overwritten, and the system will thus crash.

If enough characters are taken off the end of the line, not all the code will be overwritten. In this case, if the return instruction remains intact, the program will return to address 74, which, due to the overflow, will contain a crash opcode. In other words, this is a pretty obnoxious and certain brute force method to crash the system if there is no protection. This will even circumvent random canaries.

- readn smash: a small smash for the insecure readn operation (although it will probably also work on the insecure read) – thus the input buffer has just enough to overwrite the return address to point to the contents of the buffer, which are all the crash opcode. This is a tight and effective method for crashing the system.

- interactive smash: a simple demonstration of some of the power of overwriting memory: the return address is overwritten to point to the contents of the buffer, which contains a call to a function that will be written below the stack. After passing the return address memory cell, enough space is left for the stack frame that will be written when our rogue function is called, and thereafter the function's code is written. The function declares a variable, reads the text "i pwnd u!!!" into that variable, prints that to the screen, then crashes the system.

## 2.7 Countermeasures

Several countermeasures have been implemented that can be applied to the system to demonstrate their effectiveness at stopping attacks. The countermeasures are:

- Canaries: in the tradition of most modern solutions to buffer overflow in C compilers, canaries have been implemented as a possible countermeasure for use in this system. In each case, the canary value is checked for consistency during a return instruction, and the program is terminated if the canary is inconsistent. The following types of canaries are available:

  Terminator Canaries: simply checks that the cell after variables in the stack is still a null. While simple, it is easy for an attacker to bypass this check by ensuring a 0 gets written to that cell.

  Random Canaries: a random value is set at the beginning of program execution. For some level of realism, this value is stored in the memory cell right above the last code instruction. This is a known

location to the runtime environment (stored in a "codetop" variable). Obviously this means that if the attacker can read the memory cell containing the value or overwrite it (as in the code smash input sample), he can know what to write into the canary cell and bypass the protection.

Random XOR Canaries: The random XOR canary still sets a random canary value at the beginning of program runtime, and stores it in the cell after the last instruction. However, the value that is stored in the canary cell in the stack is the canary value XORed with the return address of the given stack frame. XORing makes it more difficult for the attacker to perform an exploit that overwrites the return address. However, if the attacker takes time to compute the data, it is far from impossible.

- Read-Only Code Segment: prevents variable writes from writing past the top of the code segment. This is a minimal countermeasure, but does provide some protection against code smashing.
- Bounds Checking: ensures that writes to variables only occur within the space allocated for said variable. This pretty much terminates all the fun, but is effective for demonstrative purposes.

# 3 Conclusion

This system provides a relatively simple simulation of buffer overflow vulnerabilities and some of their solutions for educational purposes. In addition, it's good fun to play around with.

# References

[1] Aleph One. *Smashing The Stack For Fun And Profit*

`http://insecure.org/stf/smashstack.html`