Buffer Overflows – Smashing the Stack

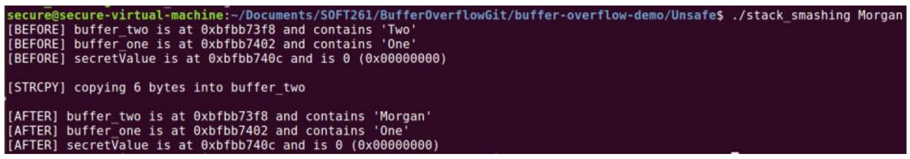
Buffer overflows are a classic security exploit well known amongst software developers. First documented in 1978 in the Computer Security Technology Planning Study, the attack vector relies on the compiler not checking the source and destination address of the next instruction. This allows for data to be overwritten or if the program counter address is changed, arbitrary code could unintentionally be executed. In a computer system, program data is written to a buffer briefly to allow transfer across program threads. They are crucial in handling communications between program threads and input/output devices. This data is stored in blocks (the size of which is determined at instantiation) side by side. It is possible for a manipulative user to exploit buffers by overrunning the boundary and altering adjacent data blocks. One type of overflow is the stack overflow (sometimes referred to as smashing the stack). I will also briefly mention heap overflows but since these are less common than their stack counterparts, they will not be the focus of this report.

The stack is where program instructions are stored in memory. A first-in-last-out data structure, the stack stores each line of code on a program as well as variables and return addresses (when executing functions). Stacks grow from low to high memory addresses (towards the heap) and all the data on the stack is assigned a specified amount of empty space to store its data. A nameInput variable, for example, may be assigned 16 bytes on the stack. This is all very effective and efficient at storing data but assumes that the developer is aware of responsible programming practices. There is no inbuilt functionality to ensure the contents of a variable or instruction pushed to the stack can fit in the allocated space.

An unsuspecting user (or malicious actor) could enter a name longer than 16 bytes and, assuming the input isn't sanitised, would result in this exploit occurring. The nameInput variable will break out of its assigned space and overflow into whatever is below it in the stack. The data below could be overridden completely and replaced with data of the attacker's choice. A particularly popular attack is to inject a piece of shellcode (code to open a root shell or perform a specific task) into the memory and fill a buffer with an address to said shellcode. The compiler will believe that this is a legitimate return address to the next function, and it will execute the arbitrary code. An attacker would not even need to find the exact location of the shell code so long as they used a NOP sled (an array of pointless instructions before the shellcode). The return address could be pointed to anywhere within the NOP sled and the executing flow will slide right down to the shellcode.

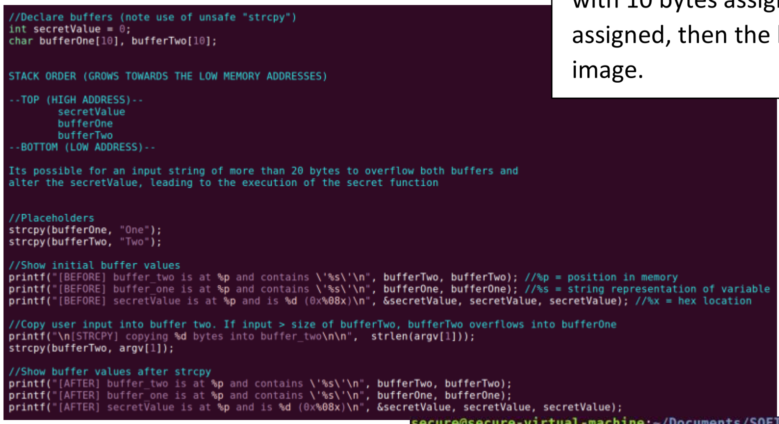
Heap based overflows, while rarer, can still occur. The heap is a dynamic data structure that grows in the opposite direction of the stack and is managed manually by the programmer (unlike the stack which the compiler of the program organises). This is where global and heap variables are often allocated by the programmer, since the heap requires variables to be allocated to it again once changes are made. Allocations are deallocated with the free() function. If this isn't done, then the heap will hold onto these variables indefinitely and cause dangerous memory leaks. These leaks are like the situation with the stack as discussed above. Using malloc() means the data is held as a persistent variable throughout the life of the program, so the neighbouring buffer to the target variable could be overflowed and the target overridden in the same way as the stack could. The difference being is that this data isn't overwritten by the compiler because the heap is audited by the developer.

For my examples and for all the references to programming material in this report, I will be using the C programming language. This is because it is, in my opinion, the easiest and infamous language to display this type of exploit in (due to its multitude of functions that have been found to be insecure to this type of attack). To test the program for yourself, see its source code and the solutions to such an exploit, please see my GitLab repository in the Appendix.



```
secure@secure-virtual-machine:~/Documents/SOFT261/BufferOverflowGit/buffer-overflow-demo/Unsafe$ ./stack_smashing Morgan
[BEFORE] buffer_two is at 0xbfbb73f8 and contains 'Two'
[BEFORE] buffer_one is at 0xbfbb7402 and contains 'One'
[BEFORE] secretValue is at 0xbfbb740c and is 0 (0x00000000)

[STRCPY] copying 6 bytes into buffer_two

[AFTER] buffer_two is at 0xbfbb73f8 and contains 'Morgan'
[AFTER] buffer_one is at 0xbfbb7402 and contains 'One'
[AFTER] secretValue is at 0xbfbb740c and is 0 (0x00000000)
```

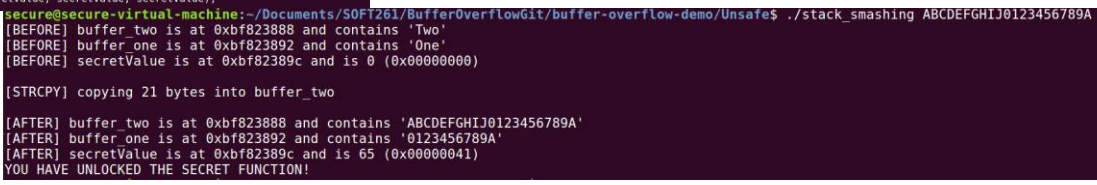
Figure 1: The program takes a string as an input and prints out the contents and location of each buffer before and after the input is copied over to the buffer. The variable secretValue is never changed within the code or by user input.



```
//Declare buffers (note use of unsafe "strcpy")
int secretValue = 0;
char bufferOne[10], bufferTwo[10];


STACK ORDER (GROWS TOWARDS THE LOW MEMORY ADDRESSES)

--TOP (HIGH ADDRESS)--
        secretValue
        bufferOne
        bufferTwo
--BOTTOM (LOW ADDRESS)--

Its possible for an input string of more than 20 bytes to overflow both buffers and
alter the secretValue, leading to the execution of the secret function


//Placeholders
strcpy(bufferOne, "One");
strcpy(bufferTwo, "Two");

//Show initial buffer values
printf("[BEFORE] buffer_two is at %p and contains \'%s\'\n", bufferTwo, bufferTwo); //%p = position in memory
printf("[BEFORE] buffer_one is at %p and contains \'%s\'\n", bufferOne, bufferOne); //%s = string representation of variable
printf("[BEFORE] secretValue is at %p and is %d (0x%08x)\n", &secretValue, secretValue, secretValue); //%x = hex location

//Copy user input into buffer two. If input > size of bufferTwo, bufferTwo overflows into bufferOne
printf("\n[STRCPY] copying %d bytes into buffer_two\n\n",  strlen(argv[1]));
strcpy(bufferTwo, argv[1]);

//Show buffer values after strcpy
printf("[AFTER] buffer_two is at %p and contains \'%s\'\n", bufferTwo, bufferTwo);
printf("[AFTER] buffer_one is at %p and contains \'%s\'\n", bufferOne, bufferOne);
printf("[AFTER] secretValue is at %p and is %d (0x%08x)\n", &secretValue, secretValue, secretValue);
```

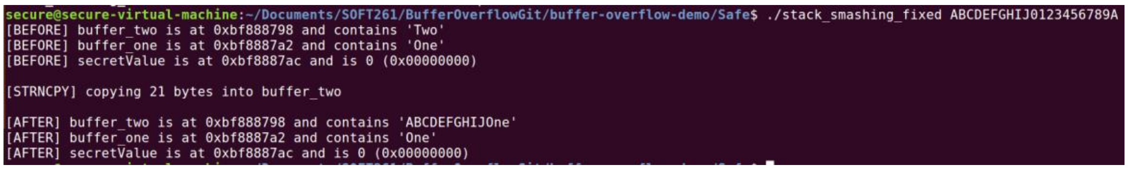
Figure 2: Buffers one and two both have been declared with 10 bytes assigned to them. If a larger value of data is assigned, then the buffers will overflow as described in the image.

Figure 3: 21 bytes are now passed into buffer two that then exceeds both it and buffer one's assigned memory. The last character then alters the secret value to 65 (ASCII value of "A"). The secret function will now be executed due to the modification of the secret value.



```
secure@secure-virtual-machine:~/Documents/SOFT261/BufferOverflowGit/buffer-overflow-demo/Unsafe$ ./stack_smashing ABCDEFGHIJ0123456789A
[BEFORE] buffer_two is at 0xbf823888 and contains 'Two'
[BEFORE] buffer_one is at 0xbf823892 and contains 'One'
[BEFORE] secretValue is at 0xbf82389c and is 0 (0x00000000)

[STRCPY] copying 21 bytes into buffer_two

[AFTER] buffer_two is at 0xbf823888 and contains 'ABCDEFGHIJ0123456789A'
[AFTER] buffer_one is at 0xbf823892 and contains '0123456789A'
[AFTER] secretValue is at 0xbf82389c and is 65 (0x00000041)
YOU HAVE UNLOCKED THE SECRET FUNCTION!
```

The primary strategy to defend against this kind of attack is to declare the maximum number of bytes the compiler should copy, this value not exceeding the buffer size allocated at variable instantiation. In C, this is represented by using the strncpy() function instead of strcpy(). The program cannot result in an overflow since the maximum buffer size will not be exceeded. Therefore, I created a 2nd program implementing this to show its validity.



```
secure@secure-virtual-machine:~/Documents/SOFT261/BufferOverflowGit/buffer-overflow-demo/Safe$ ./stack_smashing_fixed ABCDEFGHIJ0123456789A
[BEFORE] buffer_two is at 0xbf888798 and contains 'Two'
[BEFORE] buffer_one is at 0xbf8887a2 and contains 'One'
[BEFORE] secretValue is at 0xbf8887ac and is 0 (0x00000000)

[STRNCPY] copying 21 bytes into buffer_two

[AFTER] buffer_two is at 0xbf888798 and contains 'ABCDEFGHIJOne'
[AFTER] buffer_one is at 0xbf8887a2 and contains 'One'
[AFTER] secretValue is at 0xbf8887ac and is 0 (0x00000000)
```

Figure 4: Now when more than 10 bytes are passed using strncpy, the program stops the data stream and executes the next command. Shown here on my program, buffer two only contains the first 10 bytes of the user input. The secret function is not executed.

Another way of mitigating this problem is to declare any buffers in the stack before the modification sensitive variables. If the user input buffer does overflow, then it will overflow into empty memory space rather than altering sensitive data (however this is still not perfect since the program is likely to crash and an attacker could then take control of it as it does so). Some compilers include a stack protector with them (GCC included among them) that will cease the execution of a program on detection of a buffer overflow. Pseudo random values (known as canaries) are placed on the stack by the compiler after each buffer. A modified canary value will flag to the compiler that the buffer has been compromised, it will prevent the program from continuing to a potentially malicious return address.

If a programmer doesn't have a strong affinity for buffer handling or isn't aware of these issues, then perhaps a less memory centric programming language would be more suitable. C#, Java and Python don't require the memory buffer size to be specified prior to instantiation, since the memory is dynamically assigned by the compiler (these are known as type-safe languages). While this solution brings with it its own issues regarding compatibility with older systems and developer knowledge, if the programmer is familiar in a type safe language, it may be more appropriate to produce sensitive programs in this language instead.

To conclude, I believe buffer overflows to be one of the first attacks that are attempted on computer systems by security experts and malicious actors alike. The ease of testing for a vulnerable system, its high level of effectiveness if a system is vulnerable and the manoeuvrability to quickly branch off into a more devastating attack makes it very useful. While stack overflows are the most commonly tried branch of this, heap overflows are also possible, and I suspect that this type of exploit will not disappear for another 5 years at least (mostly due to high margin of human error within hundreds of lines of code).

References

- Anderson. J (1972). *Computer Security Technology Planning Study (Vol 2)* [online]. Fort Washington. Electronic Systems Division [Accessed 18/03/19]. Available from: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&ved=2ahUKEwja24-Ly4zhAhUgRxUIHQiIB8gQFjABegQIDhAC&url=http%3A%2F%2Fseclab.cs.ucdavis.edu%2Fprojects%2Fhistory%2Fpapers%2Fande72.pdf&usg=AOvVaw0_E0NzPMuOCrqNlg37un5w
- C. Cowan, F. Wagle, Calton Pu, S. Beattie and J. Walpole (2000). *Buffer overflows: attacks and defences for the vulnerability of the decade* [online]. Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00, Hilton Head, SC, USA, 2000, pp. 119-129 vol.2. Available at: https://ieeexplore.ieee.org/abstract/document/821514 [Accessed 31/03/19]
- Erickson. J (2008). *Hacking: The Art of Exploitation (2nd Edition)* [online]. San Francisco. William Pollock, No Starch Press [Accessed 26/03/19]. Available from: https://drive.google.com/file/d/1n4hcb9YoYSoVYJT2sd7UScUbxP1fd_8Z/view?usp=sharing
- Foster. J.C (2005). *Sockets, Shellcode, Porting & Coding* [online]. Andrew Williams, Syngress Publishing Inc, 800 Hingham Street, Rockland, MA [Accessed 26/03/19]. Available from: https://books.google.de/books?id=ZNI5dvBSfZoC&printsec=frontcover&q=&redir_esc=y&hl=de#v=onepage&q&f=false
- Gribble. P (2012). *Memory: Stack vs Heap* [online]. Available at: https://www.gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html [Accessed 26/03/19]
- Synopsys Editorial Team (2017). *How to detect, prevent and mitigate buffer overflow attacks* [online]. Available at: https://www.synopsys.com/blogs/software-security/detect-prevent-and-mitigate-buffer-overflow-attacks/ [Accessed 31/03/19]

Appendix

- Gitlab repository address for all the programs and code shown in the images: https://gitlab.com/M_Davies/buffer-overflow-demo