International Journal of Computer Applications (0975 – 8887)
Volume 176 – No. 39, July 2020

An Empirical Study on Stack Overflow Security Vulnerability in Well-known Open Source Software Systems

Md. Masudur Rahman
Department of CSE
Ahsanullah University of Science and Technology

Abdus Satter
Institute of Information Technology
University of Dhaka

B. M. Mainul Hossain
Institute of Information Technology
University of Dhaka

ABSTRACT

Stack overflow is one of the most common security vulnerabilities in software systems. It occurs when a program tries to load more data in a buffer than its allocated limit. It may result in serious security issue when a program having the vulnerability is run with administrator privileges. Attackers can inject malicious code into the running program through overflowing its stack. When the malicious code is executed, it allows the attackers to take control of the program. So, this security vulnerability is considered as one of the easiest and reliable techniques to gain unauthorized access to a computer system. In this article, it has been shown that how stack overflow occurs in a software system. Besides, a survey has been conducted on three popular open source projects - Linux, Git and PHP. The survey results show that the projects contain such code portions in which it is possible to overflow the stacks and inject malicious script to harm the normal execution of processes. In addition, this article raises a concern to avoid writing such codes which are potentially sources for the security attack.

Keywords

Computer Security Vulnerability, Buffer Overflow Attack, Stack Overflow, Open Source Projects, Software Security.

1. INTRODUCTION

A stack overflow (also known as buffer overflow) condition occurs when a program tries to store more data in its allocated buffer that exceeds the allowed size of the buffer. It causes the program to destroy data, change files and execute malicious code, because it overwrites the adjacent buffers in the computer memory. It happens when developers write code that does not check the boundary of input data. Due to the increased presence of stack overflow vulnerabilities in all types of software systems [14], attackers use these to access and take control of the systems [5] [20]. These vulnerabilities can occur in an Operating System (OS) or an application, or even in hardware such as network devices [17]. These are also commonly used in the remote network penetration, because attackers inject malicious code through overflowing buffers. The code may contain instructions to change the execution flow of the infected program and take control of the whole system, which ultimately results in serious security threats [6] [15].

Usually, different programming languages, for example, C, C++, etc. provide some built-in functions which are vulnerable to buffer overflow attacks [17] [18]. For instance, C programming language provides some built-in functions such as *gets()*, *strcpy()*, *strcat()*, *scanf(%s)*, etc. in which stack overflow attack could be possible.

A buffer is a contiguous block of memory which is used to store data during the execution of a program. For example, a buffer is known as an array in C programming language. Usually, two types of variables are found in a program – static and dynamic. Data segment of the buffer is used to allocate static variables at load time of the program. On the other hand, stack segment of the buffer is used to store all the dynamic variables at runtime of the program. In order to overflow the buffer, it is required to fill with data until the boundaries of the buffer are crossed. Overflowing the stack segment of the buffer is known as stack-based buffer overflow [10]. Before diving into the stack overflow vulnerability in details, it is needed to understand the memory organization of a process that has been discussed in the following subsection.

1.1 Memory Organization of a Process

To understand stack buffers, first of all, it is needed to understand how a process is organized in a memory. When a program is in execution mode, it is termed as a process [11]. A process must execute in a sequential manner. In order to accomplish a task, a process needs certain resources such as memory, CPU cycle, I/O devices, etc. Usually, a process comprises three regions – Text, Data and Stack [4] as shown in Figure 1.

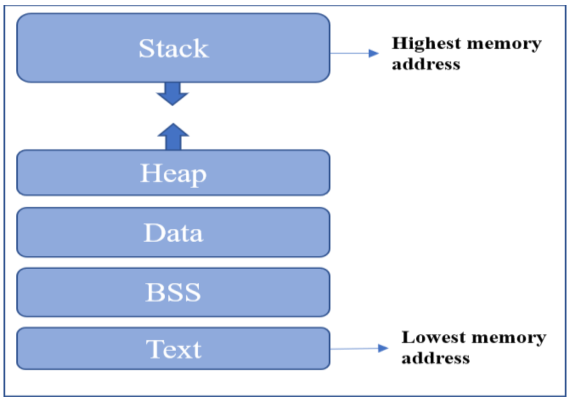


Fig 1: Process Memory Organization

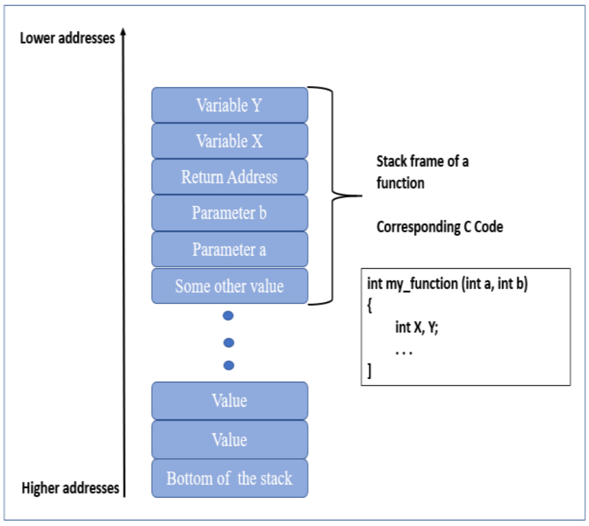
The text region of a process is the read-only portion of the process memory which is used to store code, instructions and executable files. It is also termed as code region. The BSS segment contains all the uninitialized variables of the program, and the variables are initialized with zeroes. All the static and global variables of the program are stored in the data segment. Heap is used to store dynamic variables that are declared at process runtime. Memory management module of OS usually handles the size of the heap. Stack segment of the process stores all the function calls at process runtime along with their arguments. The size of stack is dynamically managed by OS during process runtime, and it grows or shrinks based on the number of function calls and the size of the corresponding arguments.

1.2 Structure of a Stack

A stack is an important part of memory that is used to hold data. Basically, it is a contiguous block of memory containing data that resizes dynamically according to process needs. A stack has a register that points to the top of the stack. The register is known as Stack Pointer (SP). There is a bottom part of the stack which has a fixed address. Actually, SP is responsible to dynamically adjust its size with the help of the kernel at runtime of the process. It is important to note that, there are two key operations of stack: PUSH and POP. PUSH operation adds an element at the top of the stack. In contrast, POP operation reduces the stack size by one through removing the last element at the top of the stack. The Central Processing Unit (CPU) implements the instructions to PUSH onto and POP off of the stack. Specifically, a stack of elements has the property that the last element placed on the stack will be the first element to be removed. This property is commonly known as Last In, First Out queue or a LIFO.

Usually, the stack segment in a process memory grows up or down based on its implementation scheme [6]. When it grows up, it goes toward the higher memory addresses. When it grows down, it goes to the lower memory addresses. Although SP of a stack may point the last address or its immediate next available address depending on its implementation, it is assumed in the example that it points to the last address of the stack.

A stack holds a set of stack frames of a program. When a function is called, a stack frame is created. The frame contains the parameters of the function, local variables, relevant data of the previous stack and the instruction pointer's value when the function is called. Each time when a function is called, its corresponding stack frame is created and pushed into the stack. When the function returns, the corresponding stack frame is popped from the stack.



```
int my_function (int a, int b)
{
      int X, Y;
      …
]
```

Fig 2: Stack Structure of a Function

An example is demonstrated in Figure 2 to understand the memory organization of a function. As shown in the figure, the function has two parameters and two local variables. When the function is called, a stack frame is created which contains all the parameters and local variables. It also contains return address and some other related values for pointing to the previous stack frame. In the example, the stack frame is pushed into the stack from higher address to lower address.

1.3 Our Contributions

In summary, this article makes the following major contributions:

i. Show how stack overflow occurs in a program.

ii. Show stack overflow vulnerabilities in the renowned open source projects: Linux, Git and PHP System.

iii. Provide attention not to use vulnerable codes.

The remainder of this article is organized as follows: Section 2 discusses the existing works on Buffer overflow attacks. Section 3 describes buffer overflow vulnerabilities in three popular open source systems – Git, Linux and PHP. Section 4 concludes the article.

2. RELATED WORK

Buffer overflow has become one of the most common security vulnerabilities nowadays [13] [16]. Several researches have been carried out to understand the presence and impact of this vulnerability. Some researchers proposed different techniques and tools to detect and remove the vulnerability. Some significant works in this domain are discussed below.

In comparison with other security attacks, buffer overflow attacks are very common and easy to exploit [6][12]. In paper [6], Aleph One showed the process of stack overflow attacks by injecting malicious scripts in target system. The work in this article is inspired by the Aleph's work.

Crispin Cowan et al. conducted a survey on various types of buffer overflow vulnerabilities and attacks [5]. They discussed various defensive measures to mitigate buffer overflow vulnerabilities. They investigated different combinations of approaches to find and fix buffer overflow vulnerabilities in software system without changing system functionalities and performance. StackGuard is a promising defensive method for resisting buffer overflow attacks without changing system behavior or performance [5] [7] [8] [9].

Usman et al. investigated the root cause of different security threats in web-based application [12]. They highlighted that traditional software development models cannot identify different security threats rigorously. One of the main reasons is the diversity of security threats. To alleviate this problem, they proposed a model that identifies security threats from requirement analysis to product deployment phase. The model was compared with other existing models to check its effectiveness. According to their analysis, the model helps to develop more secured web application when it is used with the existing software development model.

Satter et al conducted a survey on thirty emerging websites against four common web attacks such as Man in the Middle Attack, SQL Injection, Cross Site Scripting and Denial of Service [19]. According to their study, most of the websites are vulnerable to these attacks. They found that security issues were not considered during the designing and development phase of those sites. This reveals how important to check the security issues during software development and maintenance phases.

Sariman et al. proposed a security assessment method for software systems [13]. The method can be used to detect different security vulnerabilities in the earlier stage through analyzing source code and calculating different metrices. It is found in the literature that identifying and fixing security issues earlier reduces software development time and cost significantly. Usually, security vulnerabilities are injected or created during the design phase when proper security measures are not taken care of. These vulnerabilities are reflected in the implementation phase, and thus, the final product contains security flaws. Identifying those security issues in the final product is more time consuming and expensive. The cost increases when developers need to change the source code. Sariman et al. showed that automatically identification of vulnerabilities using their proposed method significantly decreases overall software development cost and time.

Significant number of works have been carried out to understand how attackers exploit stack overflow attacks in software systems. To the best of the authors' knowledge, no one finds the vulnerabilities in the existing widely used open source projects like – Linux [1], Git [2], PHP [3] and many others. In this paper, source code of three popular open source projects – Linux, Git and PHP have been analyzed to find stack overflow vulnerabilities in the systems. Besides, vulnerable functions or code snippets in the systems in which it is possible to exploit stack overflow attacks are also described in the article. While using the systems, it has been encouraged to avoid those vulnerable codes or functions that cause stack overflows.

3. EXPERIMENTAL ANALYSIS OF STACK OVERFLOW

Stack overflow occurs when a program writes more data to a buffer located on the stack than its original size. Usually, buffers are created to store a fixed amount of data. However, when the buffer is used to store extra data that exceeds its capacity, the data is loaded into the adjacent buffers. It causes the data previously written into the adjacent buffers to be overwritten or corrupted. This situation may occur accidentally due to programming error or intentionally by attackers. When a program has the vulnerability, attackers could push malicious code in the extra data. The code may contain instructions to steal data, take control of the program, change data, damage confidential information, take full control of the system, etc. The popularity of stack overflow attacks is increasing rapidly, because many systems are built using the frameworks or libraries (specially in C and C++) that contain buffer overflow vulnerabilities in the source code. Some poor programming practices are also responsible for this.

```
void copy(char *str)
{
     char mem[25];
     strcpy(mem, str);
}

void main()
{
     int i, len = 300;
     char input_str[len+1];
     for(i=0; i<len; i++)
     {
          input_str[i]='B';
     }
     input_str[len]='\0';
     copy(input_str);
}
```

Fig 3: Example of a Typical Stack Overflow

A function with a typical buffer overflow vulnerability is shown in Figure 3. In the figure, the function *copy()* takes a string as input without any bound checking and copies it into a variable *mem[]* using *strcpy()* instead of *strncpy()*. When the program is executed, it will show segmentation fault message.

The structure of the stack is shown in Figure 4 when the function is called. Content of the variable *input_str* is assigned to the function parameter **str*. Next, the function *strcpy()* copies the content of **str* into another variable *mem[]* until a null character is encountered.

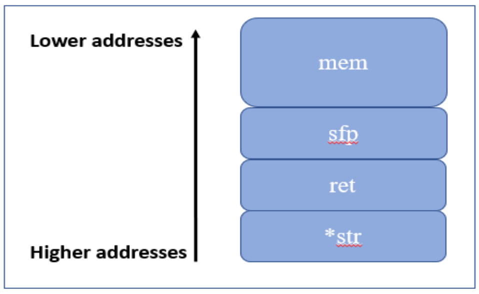


Fig 4: Stack Structure for The Function in Fig 3

It can be seen in Figure 3 that the size of *mem[]* is 25, and the size **str* is 300. The program is trying to store all the 300 bytes into *mem[]*. 1st 25 bytes will be stored into *mem[]*, that means 25 bytes of the stack allocated for *mem[]* will be used, and rest 275 bytes after *mem* of the stack will be overwritten. So, it will change the values of the Stack Frame Pointer (SFP), Return Address (RET) and **str* as shown in Figure 4. Since the *input_str* is filled with 'B' and the hex value of 'B' is $(42)_{16}$, the return address will be 0x424242. This points to the outside of the process address space, and when the function tries to return the value, it will get segmentation violation. This is the basic scheme to change the execution flow of a program through overflowing a stack.

3.1 Functions with Stack Overflow Vulnerabilities

Like *strcpy()*, C programming language have some widely used built-in functions which are vulnerable for stack overflow attacks. The standard C library provides a number of functions for copying or appending strings, that perform no boundary checking. These include: *strcat()*, *strcpy()*, *sprintf()* and *vsprintf()*. These functions operate on null- terminated strings and do not check for overflow of the receiving string. *gets()* is another example of the functions that reads a line from *stdin* into a buffer until it finds a terminating newline or End of File (EOF). It performs no checking for buffer overflows. The *scanf()* function also suffers from buffer overflow vulnerabilities due to not considering boundary checking. The reason is that when *%s* is used in the function for taking string as input, it stops taking input until it encounters a white space character in the input string.

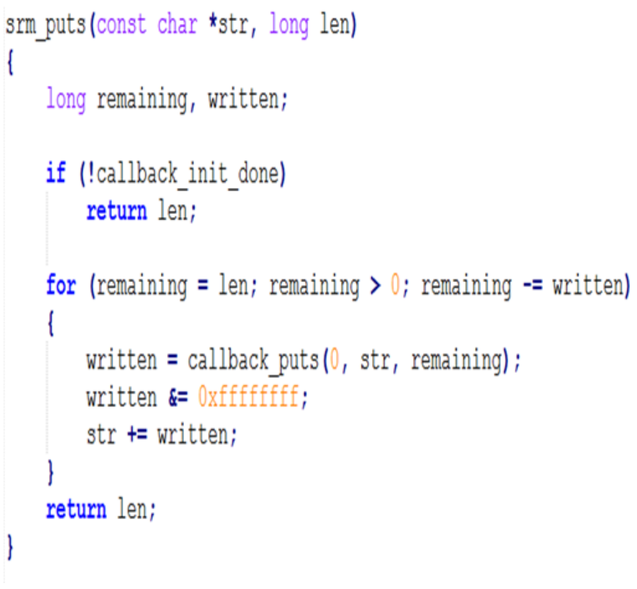
3.2 Stack Overflow Vulnerabilities in Open Source Projects

In this article, it has been shown that renowned open source projects which have been developed in C programming language, have such stack overflow security vulnerabilities. More specifically, the article presents the stack overflow vulnerability code portions in three popular open source projects - Linux [1], GIT [2] and PHP [3]. Detailed analysis of the vulnerabilities of the projects is given in the following subsections.

3.2.1 Vulnerable Codes in Linux System

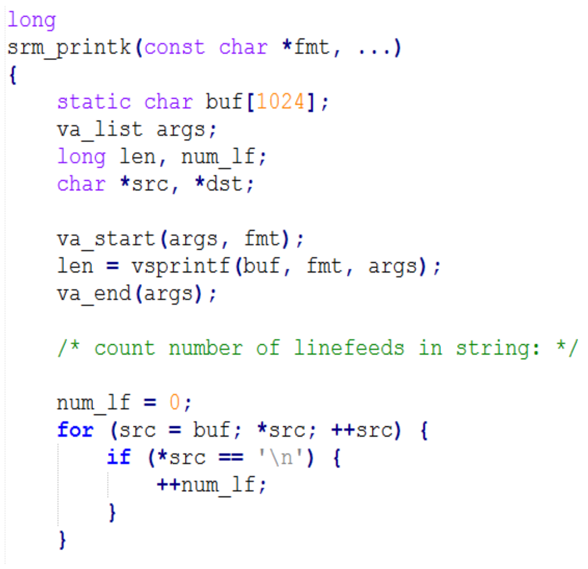
Linux is a well-known operating system and widely used by software developers or programmers. However, as this system has been developed in C, it contains several functions which are vulnerable for stack overflow attacks. As a result, an attacker can easily take control over the system through overflowing its stack or can hamper the system by injecting malicious scripts. Figure 5 shows such a vulnerable code that has been found at the file *srm_puts.c* in Linux [1], since it contains character pointer **str* as a parameter. **str* is the vulnerable portion of the code, as it is possible to call the *srm_puts()* function having a large argument for **str* parameter that results in stack overflow condition.

Some other examples are shown in Figure 6 where vulnerable code snippets are - function *vsprintf()* and parameter **fmt*. Here, the **fmt* parameter can be overflowed by providing a larger argument. In Figure 7, the parameters *buf*, *fmt* and the statement **str++ = *fmt* in the loop are also vulnerable to stack overflow attacks.



```
srm_puts(const char *str, long len)
{
    long remaining, written;

    if (!callback_init_done)
        return len;

    for (remaining = len; remaining > 0; remaining -= written)
    {
        written = callback_puts(0, str, remaining);
        written &= 0xffffffff;
        str += written;
    }
    return len;
}
```

Fig 5: Vulnerable Code in Linux: *srm_puts.c* File

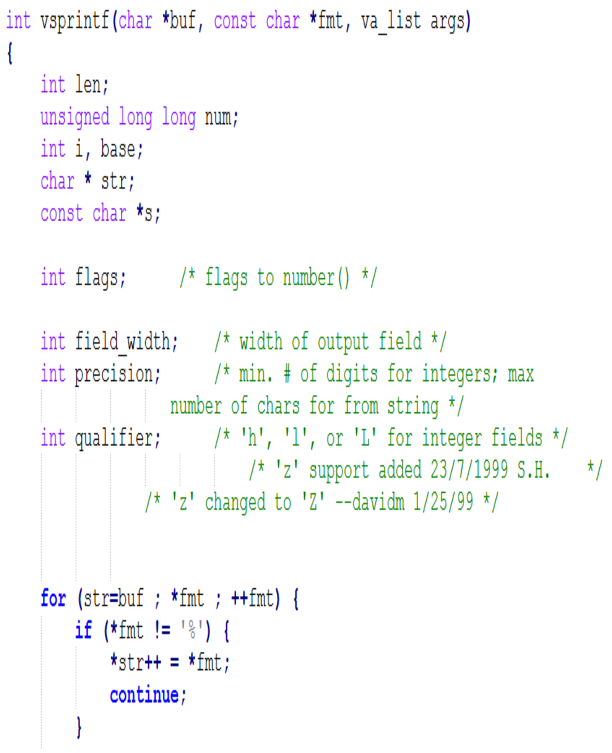


```
long
srm_printk(const char *fmt, ...)
{
    static char buf[1024];
    va_list args;
    long len, num_lf;
    char *src, *dst;

    va_start(args, fmt);
    len = vsprintf(buf, fmt, args);
    va_end(args);

    /* count number of linefeeds in string: */

    num_lf = 0;
    for (src = buf; *src; ++src) {
        if (*src == '\n') {
            ++num_lf;
        }
    }
```

Fig 6: Vulnerable Code in Linux: *srm_printk.c* File

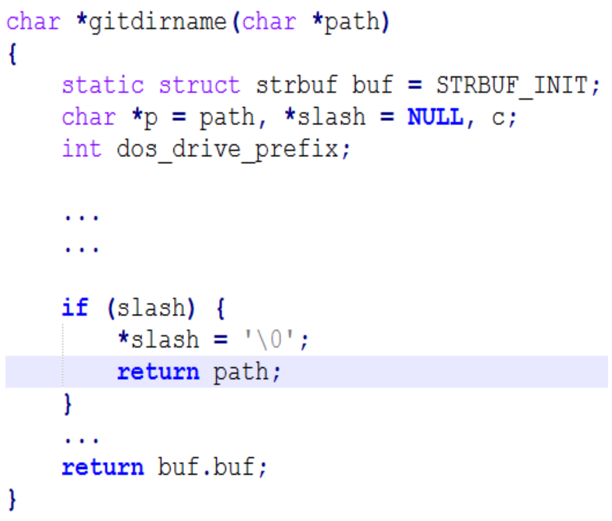
3.2.2 Vulnerable Codes in Git System

Git is a popular open source version control system [2]. It is widely used for managing source code of software system. It has been developed in C programming language, and hence some stack overflow vulnerabilities are found in the Git project [2]. These vulnerabilities are shown in Figure 8, 9 and 10.



```
int vsprintf(char *buf, const char *fmt, va_list args)
{
    int len;
    unsigned long long num;
    int i, base;
    char * str;
    const char *s;

    int flags;      /* flags to number() */

    int field_width;    /* width of output field */
    int precision;      /* min. # of digits for integers; max
                number of chars for from string */
    int qualifier;      /* 'h', 'l', or 'L' for integer fields */
                        /* 'z' support added 23/7/1999 S.H.    */
            /* 'z' changed to 'Z' --davidm 1/25/99 */


    for (str=buf ; *fmt ; ++fmt) {
        if (*fmt != '%') {
            *str++ = *fmt;
            continue;
        }
```

Fig 7: Vulnerable Code in Linux: *stdio2.c* File



```
char *gitdirname(char *path)
{
    static struct strbuf buf = STRBUF_INIT;
    char *p = path, *slash = NULL, c;
    int dos_drive_prefix;

    ...
    ...

    if (slash) {
        *slash = '\0';
        return path;
    }
    ...
    return buf.buf;
}
```

Fig 8: Vulnerable Code in Git: *basename.c* File

In Figure 8, if a malicious script is passed through the function's argument *char *path*, and this could be possible to jump to the attacker instructions through the statement return path. This could cause the severe hamper of the system. In another code snippet shown in Figure 9, it could be possible to overflow the parameter *char hashout[20]* of the function by providing an argument, that is larger than its original size. In Figure 10, the vulnerable statement is *fputs(prompt, output_fh)*, since an attacker could use the **prompt* parameter in order to inject vulnerable script that could be stored in the file located by the variable *output_fh*.

```
void blk_SHA1_Final(unsigned char hashout[20], blk_SHA_CTX *ctx)
{
    static const unsigned char pad[64] = { 0x80 };
    unsigned int padlen[2];
    int i;

    /* Pad with a binary 1 (ie 0x80), then zeroes, then length */
    padlen[0] = htonl((uint32_t)(ctx->size >> 29));
    padlen[1] = htonl((uint32_t)(ctx->size << 3));

    i = ctx->size & 63;
    blk_SHA1_Update(ctx, pad, 1 + (63 & (55 - i)));
    blk_SHA1_Update(ctx, padlen, 8);

    /* Output hash */
    for (i = 0; i < 5; i++)
        put_be32(hashout + i * 4, ctx->H[i]);
}
```

Fig 9: Vulnerable Code in Git: *sha1.c* File

```
char *git_terminal_prompt(const char *prompt, int echo)
{
    static struct strbuf buf = STRBUF_INIT;
    int r;
    FILE *input_fh, *output_fh;

    ...

    fputs(prompt, output_fh);


    r = strbuf_getline_lf(&buf, input_fh);
    if (!echo) {
        putc('\n', output_fh);
        fflush(output_fh);
    }
```

Fig 10: Vulnerable Code in Git: *terminal.c* File

3.2.3 Vulnerable Codes in PHP:

PHP is a server scripting language and a powerful platform for making dynamic and interactive Web pages. It is a widely-used open source platform developed using C programming language. However, it also contains some vulnerable codes, some of those are shown in Figure 11, 12 and 13.

An example code snippet is shown in Figure 11 where the built-in function *strcpy(buf, tmp)* in C which is severely vulnerable to stack overflow attacks, because *strcpy()* does not check the input boundary resulting the buffer overflow condition. In another example shown in Figure 12, a vulnerable function *vsprintf()* is used. In Figure 13, the return statement is vulnerable as its return control could be dominated by an attacker through the parameter **pattern* and could access the control. There exist many vulnerable codes in PHP system like the examples stated above.

```
PHPAPI char *php_ctime_r(const time_t *clock, char *buf)
{
    char *tmp;

    local_lock(CTIME_R);

    tmp = ctime(clock);
    strcpy(buf, tmp);

    local_unlock(CTIME_R);

    return buf;
}
```

Fig 11: Vulnerable Code in PHP: *reentrancy.c* File

```
php_sprintf (char*s, const char* format, ...)
{
  va_list args;
  int ret;

  va_start (args, format);
  s[0] = '\0';
  ret = vsprintf (s, format, args);
  va_end (args);
  return (ret < 0) ? -1 : ret;
}
```

Fig 12: Vulnerable Code in PHP: *php_sprintf.c* File

```
static const char *
rangematch(const char *pattern, char test, int flags)
{
    int negate, ok;
    char c, c2;

    ...
    ...

    return (ok == negate ? NULL : pattern);
}
```

Fig 13: Vulnerable Code in PHP: *fnmatch.c* File

Like those examples, there exist many vulnerable codes in Linux, Git and PHP systems. So, these vulnerable code snippets should be removed or rewritten with boundary checking, and vulnerable C built-in functions should be avoided.

4. CONCLUSION

Stack overflow is such a severe security vulnerability that an attacker could easily damage the control of the actual program flow. An attacker can easily get the control of the computer which is running the process having vulnerable codes. This article has shown the stack overflow security vulnerabilities that occur due to the use of some built-in C functions unconsciously and without boundary checking of the buffer. However, many popular open source projects like – Linux, Git, PHP, etc. contain such vulnerabilities which have been discussed in this article. This article tries to raise a concern among the developers to build applications secured from stack overflow attacks by avoiding or carefully using vulnerable built-in functions. The article suggests that developers should give extra attention when they use built-in C functions such as - *gets(), strcpy(), strcat(),* etc.

5. REFERENCES

[1] "GitHub - torvalds/linux: Linux kernel source tree", https://github.com/torvalds/linuxl, Online; accessed 06 May, 2019.

[2] "GitHub - git/git: Git Source Code Mirror", https://github.com/git/git, Online; accessed 06 May, 2019.

[3] "GitHub - php/php-src: The PHP Interpreter", https://github.com/php/ php-src, Online; accessed 06 May, 2019.

[4] Silberschatz, A., Galvin, P.B. and Gagne, G., 2009. Operating system concepts with Java. Wiley Publishing.

[5] Cowan C, Wagle F, Pu C, Beattie S, Walpole J. Buffer overflows: Attacks and defenses for the vulnerability of the decade. InProceedings DARPA Information Survivability Conference and Exposition. DISCEX'00 2000 Jan 25 (Vol. 2, pp. 119-129). IEEE..

[6] One, Aleph. "Smashing the stack for fun and profit." Phrack. vol. 7. 1996.

[7] Cowan C, Pu C, Maier D, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q, Hinton H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. InUSENIX security symposium 1998 Jan 26 (Vol. 98, pp. 63-78).

[8] Cowan, Crispin, and Calton Pu. "Survivability from a Sow's ear: The retrofit security requirement." Proceedings of the 1998 Information Survivability Workshop. 1998.

[9] Cowan, Crispin, et al. "Protecting systems from stack smashing attacks with StackGuard." Linux Expo. 1999.

[10] Litchfield, David. "Defeating the stack based buffer overflow prevention mechanism of microsoft windows 2003 server." 2003.

[11] Silberschatz, Abraham, J. L. Peterson, and P. B. Galvin. "Operating systems." Publication By John Wiley & Sons 1991.

[12] Usman S, Niaz H. Building Secure Web-Applications Using Threat Model. International Journal of Information Technology and Computer Science (IJITCS). 2018;10(3):52-62.

[13] Sariman G, Küçüksille EU. SASMEDU: Security assessment method of software in engineering education. International Journal of Information Technology and Computer Science. 2018:1-2.

[14] Luo P, Zou D, Du Y, Jin H, Liu C, Shen J. Static detection of real-world buffer overflow induced by loop. Computers & Security. 2020 Feb 1;89:101616.

[15] Khwaja AA, Murtaza M, Ahmed HF. A security feature framework for programming languages to minimize application layer vulnerabilities. Security and Privacy. 2020 Jan 1:e95.

[16] Silverstone A, inventor; Computer Protection Ip, Llc, assignee. Protecting computing devices from unauthorized access. United States patent application US 16/520,051. 2020 Jan 9.

[17] AlHarbi KN, Lin X, inventors; NORTHERN BORDERS UNIVERSITY, assignee. Preventing stack buffer overflow attacks. United States patent US 9,251,373. 2016 Feb 2.

[18] Ye T, Zhang L, Wang L, Li X. An empirical study on detecting and fixing buffer overflow bugs. In 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST) 2016 Apr 11 (pp. 91-101). IEEE.

[19] Satter A, Hossain BM. Vulnerabilities assessment of emerging web-based services in developing countries. International Journal of Information Engineering and Electronic Business. 2016 Sep 1;8(5):1.

[20] Pincus, Jonathan, and Brandon Baker. "Beyond stack smashing: Recent advances in exploiting buffer overruns." IEEE Security & Privacy 2.4 (2004): 20-27.