WIKIPEDIA

Buffer overflow

Buffer overflows (English *buffer overflow*), or - in particular - also **stack overflows** (English *, stack overflows '*) called, among the most frequent vulnerabilities in current software that u is. a. via the internet . Essentially, in the event of a buffer overflow due to errors in the program, too much data is written to a memory area that is too small for it - the buffer or stack - which overwrites memory locations after the destination memory area.

If it is not a whole block of data, but a destination of a single record, is also by a *' pointer overflow '* (by the English *pointer* , for " pointer ") spoken, indicating where the record should be written in the buffer.

contents

Dangers due to buffer overflows

A buffer overflow can cause the program to crash, corrupt data, or corrupt data structures in the runtime environment of the program. By the latter can return address of a subroutine with any data to be overwritten, whereby an attacker by transmission of any machine code may execute any commands with the privileges of the susceptible for buffer overflow process. The goal of this code is usually to give the attacker more convenient access to the system so they can use the system for their own purposes. Buffer overflows in common server and client softwareare also exploited by Internet worms.

The most sought-after target for Unix systems is the root access, which grants the attacker all access rights. However, this does not mean, as often misunderstood, that a buffer overflow that "only" results in the privileges of a "normal" user is not dangerous. Achieving the coveted root access is often much easier if you already have user rights (*privilege escalation privilege*).

Buffer overrun attacks are an important topic in computer security and network security . Not only can you try any kind of network but also locally on the system. They are usually remedied only by short-term error corrections (patches) from the manufacturer.

In addition to programming negligence, buffer overflows are primarily made possible by computer systems based on Von Neumann architecture , according to which data and program reside in the same memory. Due to this proximity to hardware, they are only a problem under assembled or compiled programming languages . Interpreted languages are usually not vulnerable , except for errors in the interpreter , since the data storage areas are always under the full control of the interpreter.

With Protected Mode introduced on the 80286 , segmentation of linear memory physically separates the program, data, and stack memories . The access protection takes place via the memory management unit of the CPU. The operating system just needs to make sure that no more memory is made available at the same time than the linear address space is large. OS / 2 was the only widespread operating system to use memory segmentation.

Programming languages

The most significant cause of buffer overruns is the use of programming languages that do not provide the ability to automatically monitor boundaries of storage areas to prevent storage areas from overflowing . This includes especially the language C , which puts the emphasis on performance (and originally simplicity of the compiler) and waives a monitoring, and the C-development C ++, Here, a programmer is sometimes forced to generate the appropriate code by hand, often deliberately or negligently omitted. The review is often implemented incorrectly, as these program parts are usually not or insufficiently tested during the program tests. In addition, the (in the case of C ++) complex language scope and the standard library make many error-prone constructs available, to which in many cases there is hardly any alternative.

The commonly used C ++ programming language offers limited possibilities for automatically checking field boundaries. As a further development of the programming language C, it adopts all the features of C, but the risk of buffer overflows when using modern means of speech (including automatic memory management) can be largely avoided. Out of habit, compatibility reasons for existing C code, system calls in C convention as well as for performance reasons, these possibilities are not always used. Run-time checks are not part of the language, unlike languages such as Pascal or Ada , but can be used in some applications (eg with smart pointers)).

Since most programming languages also define standard libraries, choosing a language usually means using the corresponding standard libraries. In the case of C and C ++, the standard library contains a number of dangerous functions, some of which do not allow safe use and some of which have no alternatives.

At the programming language level, the risk of buffer overruns can be reduced or eliminated by using programming languages that are conceptually more secure than C ++ or C. A much lower risk, for example, in programming languages of the Pascal family Modula , Object Pascal or Ada.

Buffer overflows, for example in the Java programming language, are almost excluded , since the execution is monitored in the bytecode . But Java also has buffer overflows that are caused by the runtime system and that affect several JRE versions. [1] [2] On the other hand, the Java runtime throws one `java.lang.StackOverflowError`when overflows the method call stack due to a faulty infinite recursion. This is a logical programming error of the application programmer, not the runtime system.

Processors and Programming

Other peculiarities of C and C ++ as well as the most commonly used processors make the occurrence of buffer overflows likely. The programs in these languages consist partly of subroutines . These have local variables.

In modern processors, it is common to place the return address of a subroutine and its local variables on an area called a stack . The subroutine call *first* places the return address and *then* the local variables on the stack. In modern processors such as the Intel Pentium , the stack is managed through built-in processor instructions and grows necessarily *down* . If fields or strings are used in the local variables, they usually *go up*described. If the field boundary is not checked, you can reach the return address on the stack by crossing the field and if necessary modify it intentionally.

The following program piece in C, which is often used in similar form, shows such a buffer overflow:

```
void  input_line ()
{
    char  line [ 1000 ];
    if  ( gets ( line ))      // gets gets pointer to the array, no length information
        puts ( line );      // puts writes content from line to stdout
}
```

For processors that describe the stack down, it looks like this before the call to *gets* (function of the standard library of C) (apart from any existing base pointer [3]):

Return address	
1000th character	
... ...	
3rd character	
2nd character	
1st character	← Stack Pointer

The stack grows down, the variable is overwritten upwards

gets reads a line from the input and writes the characters ab *line [0]* into the stack. It does not check the length of the line. According to the semantics of C gets *gets* only the memory address as a pointer, but no information about the available length. If you enter 1004 characters now, the last 4 bytes will overwrite the return address (assuming an address is 4 bytes here) that can be directed to a program piece within the stack. If necessary, you can enter a suitable program in the first 1000 characters .

```
00@45eA/%A@4 ... ... ... ... ... ... ... ... ... ... ... ... 0A&%
```
Input, is written by gets in the stack (1004 characters)

modified return address	
line, 1000th character	
...	
line, 5th character	third byte in the code
line, 4th character	second byte in the code
line, 3rd character	Destination of the return address, program code start
line, 2nd character	
line, 1st character	← Stack Pointer

Overwriting the return address and program code in the stack

If the program has higher privileges than the user, it can gain these privileges by exploiting the buffer overflow with a special input.

Countermeasures

Program creation

A very sustainable countermeasure is the use of type-safe programming languages and tools, such as Java or C # , in which the compliance of allocated memory areas may already be checked with the compiler when compiled into machine language , but at the latest is monitored with corresponding program code at runtime . It is imperative that changing pointer variables is only allowed under strict, restrictive rules, and it is helpful in this context if only the runtime system performs automatic garbage collection.

When creating programs, therefore, care must be taken to check all field boundaries. This is the responsibility of the programmer for outdated non-typed programming languages. However, preferably the use of programming languages that automatically monitor field boundaries should be considered, but this is not always readily possible. When using C ++, the use of C-style fields should be avoided as much as possible.

```
void  input_line ()
{
    char  line [ 1000 ];
    if  ( fgets ( line ,  sizeof ( line ),  stdin ))    // fgets checks the length
        puts ( line );   // puts writes content from line to stdout
}
```

Countermeasure: fgets checks the input length

Checking the program code

Special verification tools allow the analysis of the code and detect possible vulnerabilities. However, the field boundary verification code may be incorrect, which is often not tested.

Support by compiler

In C and C ++ a very large selection of existing programs is available. Modern compilers such as new versions of the *GNU C compiler* allow the activation of verification *code* generation during translation.

Due to their design, languages such as C do not always allow the field boundaries to be checked (example: *gets*). The compilers have to go the other way: they place space between the return address and the local variables for a random number (also called "Canary"). At program start, this number is determined, each time taking different values. For each subroutine call, the random number is written in the space provided. The required code is automatically generated by the compiler. Before exiting the program via the return address, the compiler inserts code which checks the random number for the intended value. If it has been changed, the return address is also untrustworthy. The program is aborted with a corresponding message.

Return address	
Random number barrier	
line, 1000th character	
...	
line, 3rd character	
line, 2nd character	
line, 1st character	← Stack Pointer

Countermeasure: random number barrier

In addition, some compilers can also be forced to create a *copy of* the return address below the local fields when the subprogram is called . This copy is used during the return, making buffer overflows much more difficult:

Return address	
line, 1000th character	
...	
line, 3rd character	
line, 2nd character	
line, 1st character	
Copy of the return address	← Stack Pointer

Countermeasure: Copy the return address

Compiler and Compiler Extensions

For example, for the GNU Compiler Collection, there are two common extensions that implement measures like those described above:

- The *Stack Smashing Protector* from IBM , formerly known as *ProPolice* (Homepage, English (http://www.trl.ibm.com/projects/security/ssp/)).

- The *Stack Guard* , developed at the Oregon Health & Science University , now in the Linux distribution *Immunix* , now at Novell .

Heap overflow

A *heap overflow* is a buffer overflow that occurs on the heap . Memory on the heap is allocated when programs request dynamic memory, such as via malloc () or the *new* operator in C ++ . If data is written to a buffer on the heap without checking the length and the amount of data is greater than the size of the buffer, it is written beyond the end of the buffer and a memory overflow occurs.

By *heap overflows* , can be carried out by rewriting pointers to functions arbitrary code on the computer especially when the heap is executable. For example, FreeBSD has heap protection, which is not possible here. They can only occur in programming languages in which there is no length check for buffer accesses. C , C ++ or assembler are vulnerable, Java or Perl are not.

z. For example, on June 23, 2015, Adobe announced that such a buffer overflow could execute arbitrary malicious code on systems, thereby taking control of the system on which the Flash Player was installed. [4] [5]

See also : Shellcode and Exploit

Example

```
#define BUFSIZE 128

char  *  copy_string ( const  char  * s )
{
    char  *  buf  =  malloc ( BUFSIZE );  // Assumption: Longer strings never occur

    if  ( buf )
        strcpy ( buf ,  s );  // heap overflow, if strlen (s)> 127

    return  buf ;
}
```

Since strcpy () does not check source and target sizes, but expects a zero-terminated ('\ 0') space as the source, the following variant is also unsafe (but will not overshoot "buf", but overruns if necessary) the end of the "s" allocated memory area).

```
char  *  buf ;

buf  =  malloc ( 1  +  strlen ( s ));  // Plus 1 because of the terminating NUL character
if  ( buf )
    strcpy ( buf ,  s );
```

The strncpy command on the other hand copies a maximum of n characters from the source to the destination, and thus works if s is zero-terminated or greater than BUFSIZE.

```
char  * buf ;

if  (( buf  =  malloc ( BUFSIZE ))  ! =  NULL )  {  // Check the pointer.
    strncpy ( buf ,  s ,  BUFSIZE  -  1 );
    buf [ BUFSIZE  -  1 ]  =  '\ 0' ;   // Disadvantage: The string must be terminated manually.
}
return  buf ;
```

Some operating systems, such as For example, OpenBSD provides the function *strlcpy* , which in turn ensures that the destination string is null-terminated and makes it easier to detect a truncated destination string.

See also

- attack vector
- Arithmetic overflow
- NX bit

Literature

- Aleph One: *Smashing The Stack For Fun And Profit (http://fringe.davesource.com/Fringe/Hacking/Documents/Smashing_The_Stack)* . In: *Phrack magazine* . 7, No. 49 (This article illustrates very well the operation of buffer overflows).
- Jon Erickson: *Forbidden Code.* mitp, Bonn 2004, ISBN 3-8266-1457-7 .
- Tobias Klein: *Buffer Overflows and Format String Vulnerabilities. Functioning, exploits, and countermeasures.* Dpunkt, Heidelberg 2004, ISBN 3-89864-192-9 .
- Felix Lindner: *A lot of risk. Buffer overflows on the heap and how to exploit them.* In: *c't . Magazine for computer technology.* 23, 9, 2006, p. 186-192, also online for free at heise Security (https://www.heise.de/security/artikel/Ein-Haufen-Risiko-270800.html) .
- Oliver Müller: *defector. System collapse via stack overflow.* In: *iX - Magazine for Professional Information Technology* , 2, 2007, pp. 100-105.
- Stephan Kallnik, Daniel Pape, Daniel Schröter, Stefan Strobel, Daniel Bachfeld: *Pierced. Buffer overflows and other predetermined breaking points* . (https://www.heise.de/security/artikel/Eingelocht-270148.html)heise Security, also in *c't* 23/2001, p. 216 (https://web.archive.org/web/20011114032312/http://www.heise.de/ct/01/23/216/) (Memento of November 14, 2001 in the *Internet Archive*) . Introductory article with simple examples in C.

Web links

- Technical Details of Buffer Overflow (and Exploits) (http://insecure.org/stf/smashstack.html)
- Exploiting a heap overflow (https://www.heise.de/security/artikel/Ein-Haufen-Risiko-270800.html/0)
- Buffer Overflows and How to Protect Themselves (https://web.archive.org/web/20011114032312/http://www.heise.de/ct/01/23/216/) (Memento, November 14, 2001 in the *Internet Archive*)
- Buffer overflows and other predetermined breaking points (https://www.heise.de/security/artikel/Eingelocht-270148.html)

Individual proofs

1. *Vulnerability in the Sun Java Runtime Environment* (http://www.linux-community.de/Neues/story?storyid=21756) - *Linux Community* , January 17, 2007
2. Sun Java JRE to 1.5.x GIF image handler buffer overflow (http://www.scip.ch/?vuldb.2842) - *vuldb.com* , on January 22, 2007 (last modified July 7, 2015)
3. *What is a stack? (http://www.a-m-i.de/tips/stack/stack.php)*, June 2nd, 2012
4. Dennis Schirrmacher: *Adobe: Emergency Update for Flash Player.* (https://www.heise.de/security/meldung/Adobe-Notfall-Update-fuer-Flash-Player-2724360.html)In: *Heise Security.* Heise online , June 24, 2015, accessed on June 29, 2015 .
5. *Security updates available for Adobe Flash Player - CVE number: CVE-2015-3113.* (https://helpx.adobe.com/security/products/flash-player/apsb15-14.html)In: *Adobe Security Bulletin.* Adobe Systems , June 23, 2015, accessed June 29, 2015 .