Secure Programming (06-20010)
Chapter 7: Overflows

Christophe Petit

University of Birmingham

Lectures Content (tentative)

1. Introduction
2. General principles
3. Code injection (SQL, XSS, Command)
4. HTTP sessions
5. Unix Access Control Mechanisms
6. Race conditions
7. Integer and buffer overflows
8. Code review

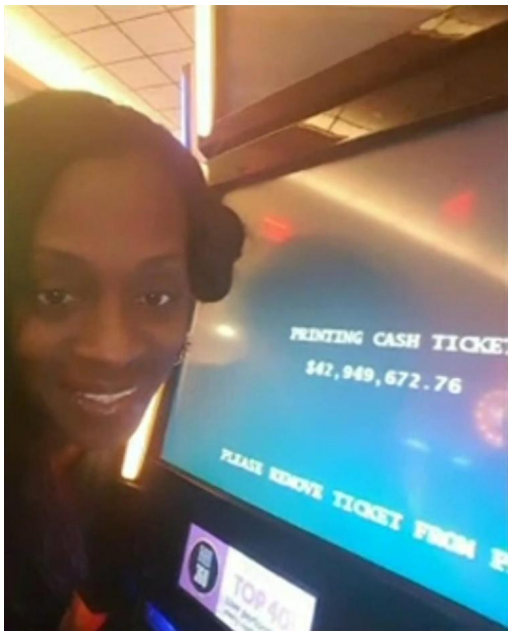
Overflows

Picture sources :
`fineartamerica.com/featured/train-derailment-at-montparnasse-station-1895-war-is-hell-store.html`, `www.oddee.com/item_98637.aspx`

Overflows (2)

Picture sources :
en.wikipedia.org/wiki/Year_2000_problem
hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/

Overflows (3)



Picture sources : www.oddee.com/item_98637.aspx
www.nydailynews.com/news/national/woman-sues-casino-offered-steak-43-million-article-1.3253502

CWE Top-25

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere
[17]	65.5	CWE-732	Incorrect Permission Assignment for Critical Resource
[18]	64.6	CWE-676	Use of Potentially Dangerous Function
[19]	64.1	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
[20]	62.4	CWE-131	Incorrect Calculation of Buffer Size
[21]	61.5	CWE-307	Improper Restriction of Excessive Authentication Attempts
[22]	61.1	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
[23]	61.0	CWE-134	Uncontrolled Format String
[24]	60.3	CWE-190	Integer Overflow or Wraparound
[25]	59.9	CWE-759	Use of a One-Way Hash without a Salt

Outline

Integer Overflows

Buffer overflows

Format String Attacks

Summary

Outline

CWE-190

CWE-190: Integer Overflow or Wraparound

Weakness ID: 190 **Status:** Incomplete
Abstraction: Base

Presentation Filter: Basic

Description

Description Summary

The software performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control.

Extended Description

An integer overflow or wraparound occurs when an integer value is incremented to a value that is too large to store in the associated representation. When this occurs, the value may wrap to become a very small or negative number. While this may be intended behavior in circumstances that rely on wrapping, it can have security consequences if the wrap is unexpected. This is especially the case if the integer overflow can be triggered using user-supplied inputs. This becomes security-critical when the result is used to control looping, make a security decision, or determine the offset or size in behaviors such as memory allocation, copying, concatenation, etc.

Outline

First example

- Consider the following C code

```
#include<stdio.h>
#include <stdlib.h>

int main() {
    int account_balance = 10000;
    printf("Your current balance is %i\n",account_balance);
    printf("How much would you like to withdraw?\n");
    char response[20];
    fgets(response, 20, stdin);
    int withdraw_amount = atoi(response);
    account_balance-=withdraw_amount;
    printf("You have withdrawn %u\n",withdraw_amount);
    printf("Your current balance is %i\n",account_balance);
}
```

- What happens if you withdraw 2,500,000,000 ?

  (for 32-bit integers)

Integer Overflows

- Data types in C have fixed sizes
- Hence they have minimum and maximum values
- Results of arithmetic operations may exceed the bounds
- Result will then typically "wrap around"
- In Maths language : operations are not over integers, but "modulo 2^{32}" (for 32-bit integers)

Data type sizes in C

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Related bugs

- Integer Truncation : most significant bits are lost when integer assigned/cast to shorter integer type
- Casting between signed/unsigned integers : same 32 bits mean either -1 or 4,294,967,295
- Sign Extension : signed integer of a smaller bit length is cast to an integer type of a larger bit length

Example : absolute value

```
#include<stdio.h>
#include<stdlib.h>

int main(){
    printf("Printing absolute value %i.\n",abs(-2147483648));
    return 0;
}
```

- abs should turn negative numbers into positive ones
- What number will be printed by the program ?
- We have $-(-2^{31}) = -2^{31}$

Example : buffer length calculation

```
nresp = packet_get_int();
if (nresp > 0) {
        response = xmalloc(nresp*sizeof(char*));
        for (i = 0; i < nresp; i++)
                response[i] = packet_get_string(NULL);
}
```

Source : OpenSSH 3.3 (Example 2 in CWE-190)

Example : casting

```
#include<stdio.h>
#include<string.h>

void bad_function(char *input){
    char dest_buffer[32];
    char input_len = strlen(input);

    if (input_len < 32)
    {
        strcpy(dest_buffer, input);
        printf("The first command line argument is %s.\n", dest_buffer);
    }
    else {
        printf("Error ? input is too long for buffer.\n");
    }
}

int main(int argc, char *argv[]){
    if (argc > 1){
        bad_function(argv[1]);
    } else {
        printf("No command line argument was given.\n");
    }
    return 0;
}
```

Code source : projects.webappsec.org/w/page/13246946/Integer%20Overflows

Example : infinite loop

```
short int bytesRec = 0;
char buf[SOMEBIGNUM];

while(bytesRec < MAXGET) {
        bytesRec += getFromInput(buf+bytesRec);
}
```

Code source : Example 3 in CWE-190

Example : Java

```
public class OverflowTest
{
    public static void main(String[] args)
    {
        int a = Integer.MAX_VALUE;
        int b = 1;

        int c = a + b;
        System.out.println(a + " + " + b + " = " + c);
    }
}
```

Integer Overflows in Various Languages

- Defined or undefined behaviour
- Exceptions thrown
- Automatic conversion to longer integers

Language	**Unsigned integer**	**Signed integer**
C/C++	modulo power of two	undefined behavior
Java		modulo power of two
Python 2		convert to long type (bigint)

Outline

Carry and Overflow tags

- Most processors have two dedicated processor flags to check for overflow conditions
- Carry flag when addition/substraction does not fit bitsize
- Overflow flag when sign different than expected
- Flags accessible in Assembly code but not in C

Listen to your compiler

- Compilers are usually not able to fix your code
- But they can output additional warnings
- Examples
  - gcc -Wsign-compare
  - gcc -Wall
  - clang -Weverything

`-Wsign-compare`

Warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. In C++, this warning is also enabled by `-Wall`. In C, it is also enabled by `-Wextra`.

`-Wsign-conversion`

Warn for implicit conversions that may change the sign of an integer value, like assigning a signed integer expression to an unsigned integer variable. An explicit cast silences the warning. In C, this option is enabled also by `-Wconversion`.

Secure integer arithmetic

- Additions : prior to computing $a + b$, check whether $a > INT\_MAX - b$
- Multiplications : write $a = a_0 + Ba_1 + B^2a_2 + \ldots$ where $B = 2^{32}$, and perform large arithmetic operations using multiple operations on words
- See `www.fefe.de/intof.html`
- Can be tricky to implement ; better to use libraries

Libraries : SafeInt

SafeInt Library

Visual Studio 2015 | Other Versions

The new home for Visual Studio documentation is Visual Studio 2017 Documentation on docs.microsoft.com.

The SafeInt library helps prevent integer overflows that might result when the application performs mathematical operations.

In This Section

Section	Description
SafeInt Class	This class protects against integer overflows.
SafeInt Functions	Functions that can be used without creating a `SafeInt` object.
SafeIntException Class	A class of exceptions related to the `SafeInt` class.

Libraries : safe-iop

safe-iop

Safe Integer Operation Library for C

This library provides a collection of (macro-based) functions for performing safe integer operations across platform and architecture with a straightforward API.

It supports two modes of use: header-only and linked dynamic library. The linked, dynamic library supplies a format-string based interface which is in pre-alpha. The header-only mode supplies integer and sign overflow and underflow pre-condition checks using checks derived from the CERT secure coding guide. The checks do not rely on twos complement arithmetic and should not at any point perform an arithmetic operations that may overflow. It also performs basic type agreement checks to ensure that the macros are being used (somewhat) correctly.

(**Note**, if you are using a version older than 0.3.1, please **upgrade**. 0.3.0 (and possibly earlier versions) will fail unnecessarily on negative addition cases.)

Project Information

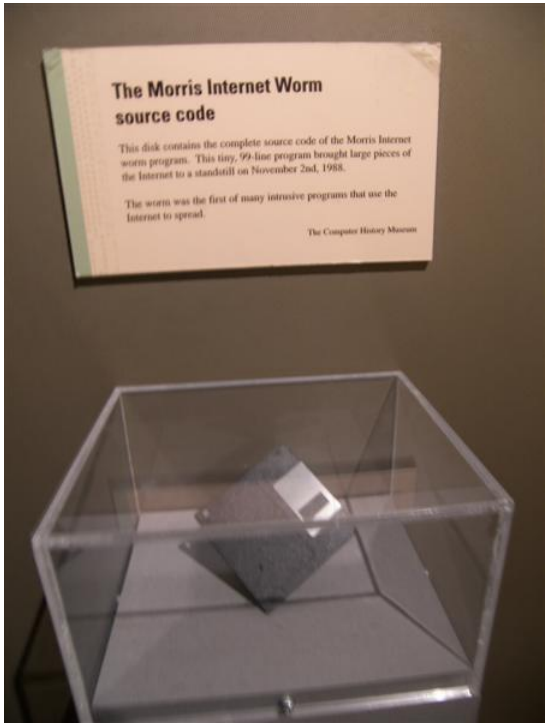
- License: New BSD License
- 6 stars
- svn-based source control

Labels:

integer security overflow safeintegeroperations operations arithmetic math library header C

Outline

Motivation


The Morris Internet Worm
source code
This disk contains the complete source code of the Morris Internet worm program. This tiny, 99-line program brought large pieces of the Internet to a standstill on November 2nd, 1988.
The worm was the first of many intrusive programs that use the Internet to spread.
The Computer History Museum


HELLO! - Netscape
File Edit View Go Communicator Help
Welcome to http://www.worm.com !
Hacked By Chinese!

Motivation : CWE Top-25

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere
[17]	65.5	CWE-732	Incorrect Permission Assignment for Critical Resource
[18]	64.6	CWE-676	Use of Potentially Dangerous Function
[19]	64.1	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
[20]	62.4	CWE-131	Incorrect Calculation of Buffer Size
[21]	61.5	CWE-307	Improper Restriction of Excessive Authentication Attempts
[22]	61.1	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
[23]	61.0	CWE-134	Uncontrolled Format String
[24]	60.3	CWE-190	Integer Overflow or Wraparound
[25]	59.9	CWE-759	Use of a One-Way Hash without a Salt

Outline

Buffer overflows : in a nutshell

- Program copies an input buffer to an output buffer

```
#include<stdio.h>

int main() {
    char last_name[20];
    printf ("Enter your last name: ");
    scanf ("%s", last_name);
}
```

- Overflow occurs if input buffer is larger than output buffer
- Can cause program crash "segmentation error" ...
- ...or execution of arbitrary code

Another example

```
#include <stdio.h>
int main(int argv, char ** argc) {
        char * names[] = {"Brazil", "Belgium", "Gibraltar"};
        int n;
        printf("Which country has the best socker team?");
        printf("\n1) Brazil\n2) Belgium\n3) Gibraltar\n");
        scanf("%d", &n);
        printf("OK, %s has the best team!\n", names[n-1]);
}
```

Example (Wikipedia)

```
char A[8] = "";
unsigned short B     = 1979;
```

variable name	A								B	
value	[null string]								1979	
hex value	00	00	00	00	00	00	00	00	07	BB

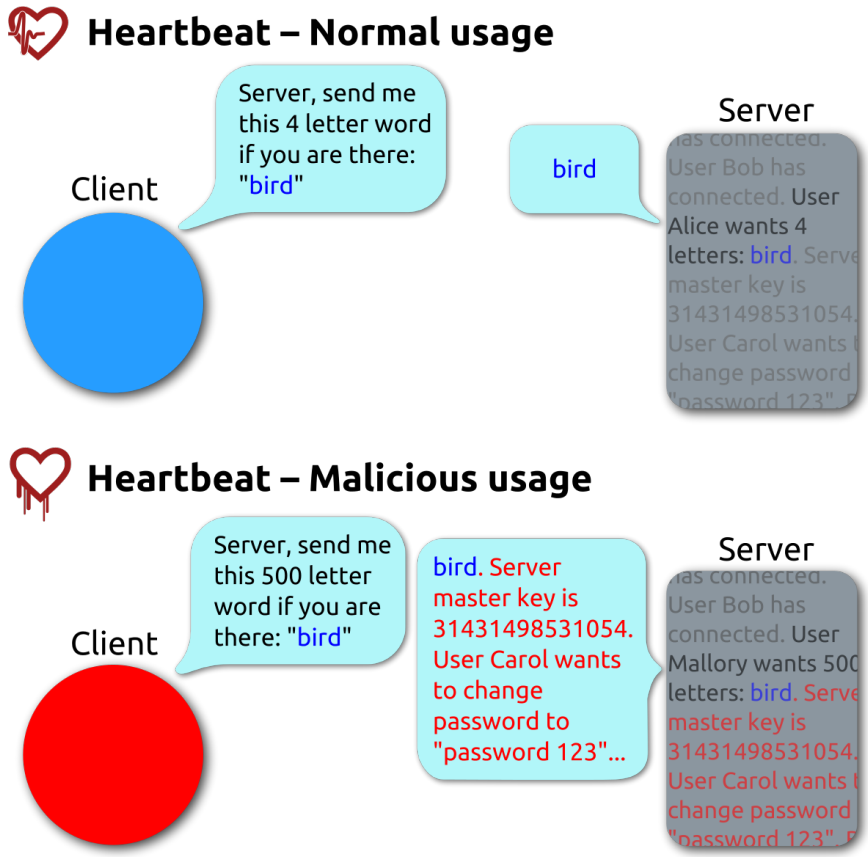
```
strcpy(A, "excessive");
```

variable name	A								B	
value	'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	25856	
hex	65	78	63	65	73	73	69	76	65	00

Integer and buffer overflows

```
#include<stdio.h>
#include<string.h>

void bad_function(char *input){
    char dest_buffer[32];
    char input_len = strlen(input);

    if (input_len < 32)
    {
        strcpy(dest_buffer, input);
        printf("The first command line argument is %s.\n", dest_buffer);
    }
    else {
        printf("Error ? input is too long for buffer.\n");
    }
}

int main(int argc, char *argv[]){
    if (argc > 1){
        bad_function(argv[1]);
    } else {
        printf("No command line argument was given.\n");
    }
    return 0;
}
```

Buffer overread : Heartbleed



Picture source : Wikipedia

Outline

Static memory allocation

- Default method for variables with file scope

```
int a = 43;
```

- Also when using static keyword

```
static int a = 43;
```

- Allocated at compile time
- Memory blocks typically along executable code
- Lifetime is the lifetime of the program

Dynamic memory allocation

- Sometimes called "heap" memory
- Allocated at runtime
- In C : use functions malloc, calloc, realloc

```
int *array = malloc(10 * sizeof(int));
if (array == NULL) {
        fprintf(stderr, "malloc failed\n");
        return(-1);
}
```

- Memory remains allocated until free is called
- In C++ : use new and delete

Dynamic memory allocation (2)

- Memory requests satisfied by allocating portions from a large pool of memory called heap or free store
- Heap typically contains chunks of memory of fixed length, or a few fixed lengths

Malloc, free, calloc, realloc

The **malloc**() function allocates *size* bytes and returns a pointer to the allocated memory. *The memory is not initialized*. If *size* is 0, then **malloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

The **free**() function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc**(), **calloc**(), or **realloc**(). Otherwise, or if *free(ptr)* has already been called before, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

The **calloc**() function allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero. If *nmemb* or *size* is 0, then **calloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

The **realloc**() function changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will *not* be initialized. If *ptr* is NULL, then the call is equivalent to *malloc(size)*, for all values of *size*; if *size* is equal to zero, and *ptr* is not NULL, then the call is equivalent to *free(ptr)*. Unless *ptr* is NULL, it must have been returned by an earlier call to **malloc**(), **calloc**(), or **realloc**(). If the area pointed to was moved, a *free(ptr)* is done.

Automatic memory allocation

- Commonly known as "stack" memory
- Typically faster than dynamic memory allocation
- Allocated at runtime when you enter a new scope (function, loop, ...)

```
int a = 43;
```

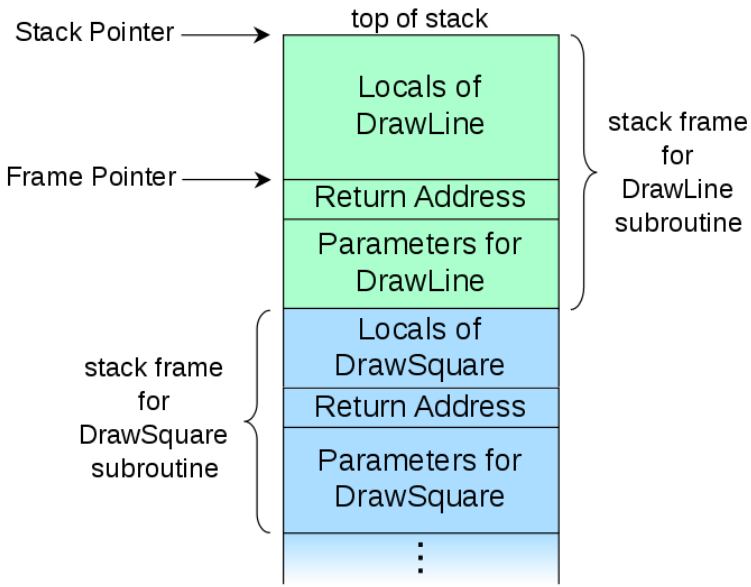
- Once you move out of the scope, values of automatic memory addresses are undefined

Call stack

- Stack : data structure with "last in, first out" access
- Call stack is a stack associated to a process
- Keeps information on all active routines
  - Function parameters
  - Local variables
  - Return address : address of instruction to execute after current routine terminates
- This information is added at the top of the stack when entering a subroutine, and removed when leaving it
- Allows efficient recursion
- Stack manipulations done for you (except in Assembly)

Call Stack : Wikipedia Example

- Function DrawSquare calling another function Drawline

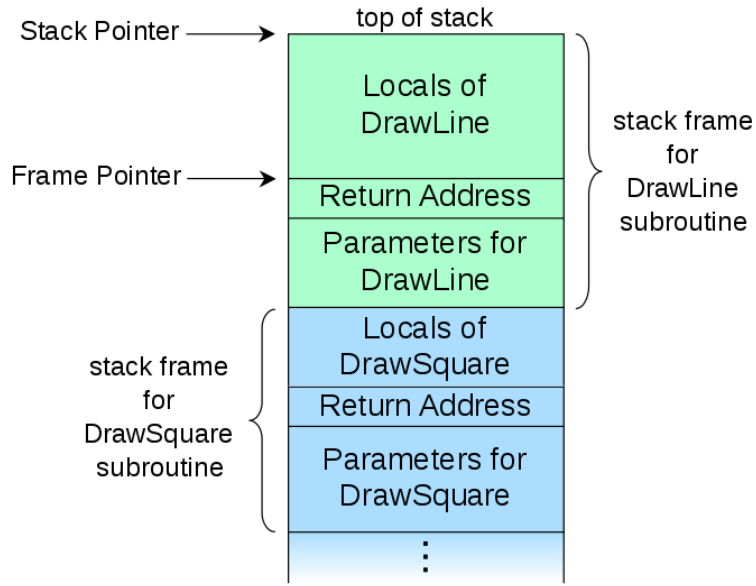


Picture source : Wikipedia

Outline

Stack Smashing with Buffer Overflows

- Overwrite local variable
  to change program behavior
- Overwrite return address
  in stack frame
- Overwrite function pointer
- Overwrite local variable
  or pointer
  in another stack frame

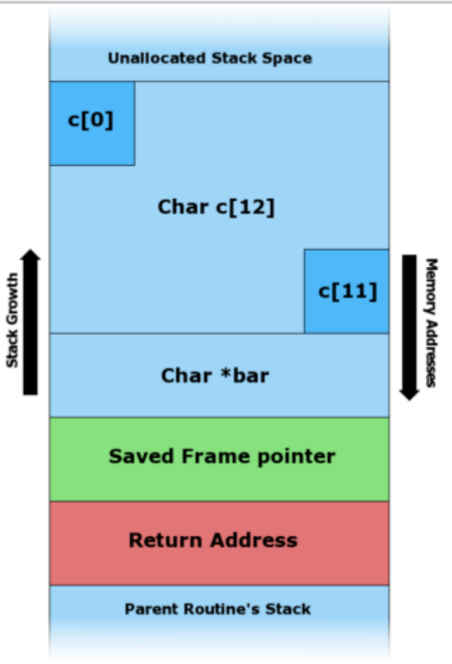


Picture source : Wikipedia

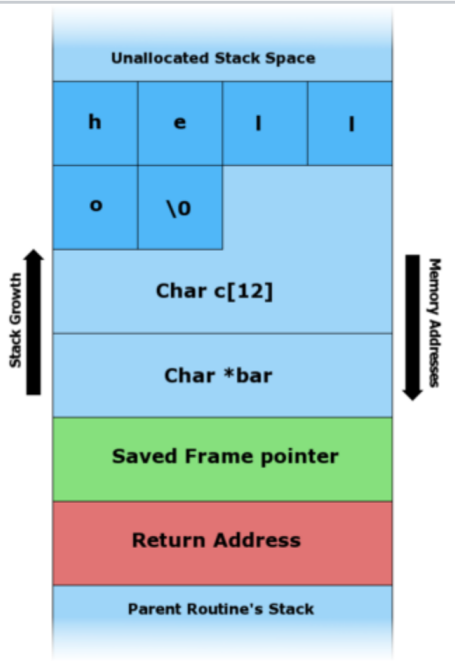
Stack Buffer Overflow Example (Wikipedia)

```
#include <string.h>

void foo (char *bar) {
    char  c[12];
    strcpy(c, bar);  // no bounds checking
}

int main (int argc, char **argv) {
    foo(argv[1]);
    return 0;
}
```

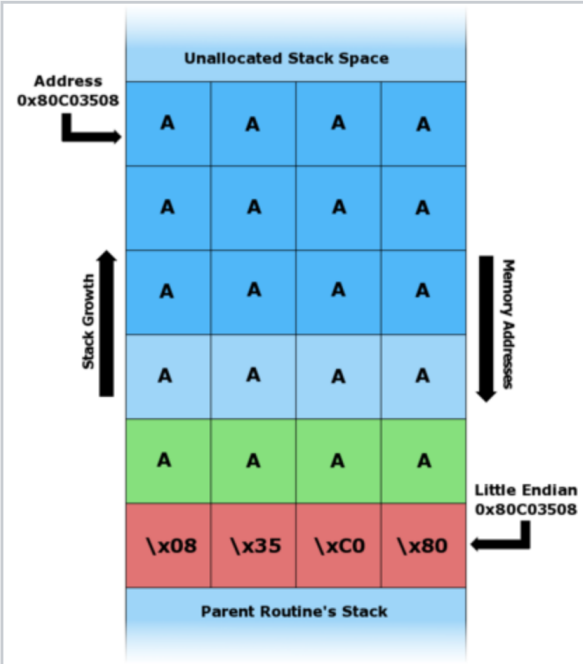
Stack Buffer Overflow Example (Wikipedia)



A. - Before data is copied.



B. - "hello" is the first command line argument.



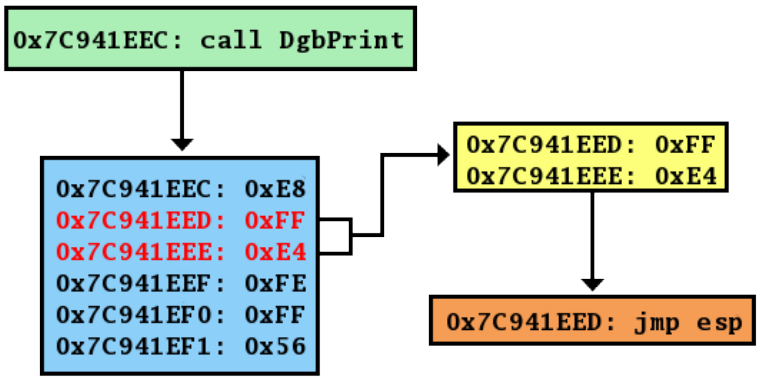
C. - "AAAAAAAAAAAAAAAAAAA A\x08\x35\xC0\x80" is the first command line argument.

NOP-slide

- Goal : replace return address by ShellCode address
- However, you don't know this address precisely (debuggers help, but addresses vary a bit)
- Idea : inject
  - Many instructions with no effect (no-ops or NOPs)
  - Followed by ShellCode
  - Followed by arbitrary content
  - Followed by approximate ShellCode address on the return address position
- Program will follow return address to the NOP zone, then "slide" on the NOPs until reaching the Shell Code

Jump to register technique

- Goal : cause a jump to malicious code



Picture credit : Wikipedia

- Wiki example :
  - "Jump to stack pointer" instruction is involuntarily present in the code
  - Attacker overwrites return address so that this instruction is executed
  - Program jumps to the top of the stack and executes the attacker's code

Other exploitation techniques

- Heap-based exploitation : typically overwrites pointer to program function
- Heap spraying : fill in the heap with large memory chunks to predict location hence facilitate exploitation
- Return programming : see presentations

Outline

Protection and prevention mechanisms

- Hardware and Operating System features
  - NX bits, W^X protections
  - Address Space Layout Randomization
- Design phase
  - Language choice
  - Protected libraries
- Implementation phase
  - Check input bounds
  - Avoid dangerous functions
- Compiler features such as Canaries
- Static code analysis

Data execution protections

- NX (No-eXecute) bits : used by CPU to distinguish data and code memory
- W^X (Write XOR eXecute) : memory pages marked by Operating System as either writable or executable, but not both

Address space layout randomization (ASLR)

- Idea : randomize address space positions including positions of the stack, heap and libraries
- Makes buffer overflow exploitation harder
- Partially defeated by NOP techniques if entropy too small

Safe vs unsafe languages

- Assembly and C/C++ particularly vulnerable
  - Direct access to memory
  - Not strongly typed
- Java, Python automatically check bounds
  so they are naturally protected
- Most scripting languages protected, throw errors

Library solutions

- No bound check with *strcpy*, *gets*, *sprintf*
- *strncpy*, *fgets*, *snprintf* take a bound argument (but beware of subtleties - see David Wheeler 6.2)
- OpenBSD : *strlcpy* and *strlcat*
- Better String, Vstr, Safestr libraries

strcpy and strncpy

strncpy(3) - Linux man page

Name

strcpy, strncpy - copy a string

Synopsis

```
#include <string.h>

char *strcpy(char *dest, const char *src);

char *strncpy(char *dest, const char *src, size_t n);
```

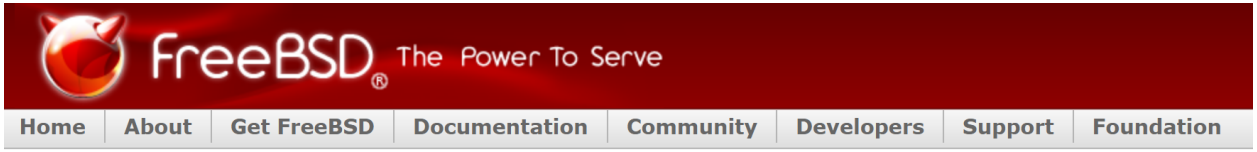
Description

The **strcpy**() function copies the string pointed to by *src*, including the terminating null byte ('\0'), to the buffer pointed to by *dest*. The strings may not overlap, and the destination string *dest* must be large enough to receive the copy. *Beware of buffer overruns!* (See BUGS.)

The **strncpy**() function is similar, except that at most *n* bytes of *src* are copied. **Warning**: If there is no null byte among the first *n* bytes of *src*, the string placed in *dest* will not be null-terminated.

If the length of *src* is less than *n*, **strncpy**() writes additional null bytes to *dest* to ensure that a total of *n* bytes are written.

strlcpy



Home | About | Get FreeBSD | Documentation | Community | Developers | Support | Foundation

FreeBSD Manual Pages

strlcpy man apropos

3 - Subroutines ▾ FreeBSD 11.1-RELEASE and Ports ▾ All Architectures ▾ html ▾

home | help

```
STRLCPY(3)            FreeBSD Library Functions Manual            STRLCPY(3)

NAME
     strlcpy, strlcat -- size-bounded string copying and concatenation

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <string.h>

     size_t
     strlcpy(char * restrict dst, const char * restrict src, size_t dstsize);

     size_t
     strlcat(char * restrict dst, const char * restrict src, size_t dstsize);

DESCRIPTION
     The strlcpy() and strlcat() functions copy and concatenate strings with
     the same input parameters and output result as snprintf(3).  They are
     designed to be safer, more consistent, and less error prone replacements
     for the easily misused functions strncpy(3) and strncat(3).

     strlcpy() and strlcat() take the full size of the destination buffer and
     guarantee NUL-termination if there is room.  Note that room for the NUL
     should be included in dstsize.
```

Better String Library

The Better String Library

by Paul Hsieh
Last updated: 07/27/2015 10:19:47

Support this project

GIT - src
08/27/2015

GIT - zip
08/27/2015

EXAMPLES
09/28/2006

Documentation
bstring.txt

Security Statement
security.txt

Introduction

The ***Better String Library*** is an abstraction of a string data type which is superior to the C library char buffer string type, or C++'s std::string. Among the features achieved are:

- Substantial mitigation of buffer overflow/overrun problems and other failures that result from erroneous usage of the common C string library functions
- Significantly simplified string manipulation
- High performance interoperability with other source/libraries which expect '\0' terminated char buffers
- Improved overall performance of common string operations
- Functional equivalency with other more modern languages

The library is totally stand alone, portable (known to work with gcc/g++, MSVC++, Intel C++, WATCOM C/C++, Turbo C, Borland C++, IBM's native CC compiler on Windows, Linux and Mac OS X), high performance, easy to use and is not part of some other collection of data structures. Even the file I/O functions are totally abstracted (so that other stream-like mechanisms, like sockets, can be used.) Nevertheless, it is adequate as a complete replacement of the C string library for string manipulation in any C program.

The library includes a robust C++ wrapper that uses overloaded operators, rich constructors, exceptions, stream I/O and STL to make the CBString struct a natural and powerful string abstraction with more functionality and higher performance than std::string.

Bstrlib is stable, well tested and suitable for any software production environment.

Canaries

- Goal : detect buffer overflow and abort the program
- Idea : include random value next to buffer, and check whether the value gets modified

Picture source : www.academia.dk/Blog/a-canary-in-a-coal-mine-in-the-19th-century-and/

Canaries : your compiler helps

- gcc -fstack-protector

`-fstack-protector`

Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call `alloca`, and functions with buffers larger than 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits.

`-fstack-protector-all`

Like `-fstack-protector` except that all functions are protected.

`-fstack-protector-strong`

Like `-fstack-protector` but includes additional functions to be protected — those that have local array definitions, or have references to local frame addresses.

- gcc -D_FORTIFY_SOURCE=2
- gcc -Wall, clang -Weverything

Testing tools

- OWASP recommendations :

Testing Buffer Overflow

- OllyDbg - http://www.ollydbg.de
  - "A windows based debugger used for analyzing buffer overflow vulnerabilities"
- Spike - http://www.immunitysec.com/downloads/SPIKE2.9.tgz
  - A fuzzer framework that can be used to explore vulnerabilities and perform length testing
- Brute Force Binary Tester (BFB) - http://bfbtester.sourceforge.net
  - A proactive binary checker
- Metasploit - http://www.metasploit.com/
  - A rapid exploit development and Testing frame work

- Also Splint, Valgrind, CBMC
- For an (old) comparison of some static analysis tools see *Testing static analysis tools using exploitable buffer overflows from open source code*, Zitser-Lippmann-Leek

Valgrind

Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

The Valgrind distribution currently includes six production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and a heap profiler. It also includes three experimental tools: a stack/global array overrun detector, a second heap profiler that examines how heap blocks are used, and a SimPoint basic block vector generator. It runs on the following platforms: X86/Linux, AMD64/Linux, ARM/Linux, ARM64/Linux, PPC32/Linux, PPC64/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, X86/Solaris, AMD64/Solaris, ARM/Android (2.3.x and later), ARM64/Android, X86/Android (4.0 and later), MIPS32/Android, X86/Darwin and AMD64/Darwin (Mac OS X 10.12).

Valgrind is Open Source / Free Software, and is freely available under the GNU General Public License, version 2.

CBMC

About CBMC

CBMC is a Bounded Model Checker for C and C++ programs. It supports C89, C99, most of C11 and most compiler extensions provided by gcc and Visual Studio. It also supports SystemC using Scoot. We have recently added experimental support for Java Bytecode.

CBMC verifies array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions. Furthermore, it can check C and C++ for consistency with other languages, such as Verilog. The verification is performed by unwinding the loops in the program and passing the resulting equation to a decision procedure.

While CBMC is aimed for embedded software, it also supports dynamic memory allocation using `malloc` and `new`. For questions about CBMC, contact Daniel Kroening.

CBMC is available for most flavours of Linux (pre-packaged on Debian, Ubuntu and Fedora), Solaris 11, Windows and MacOS X. You should also read the CBMC license.

CBMC comes with a built-in solver for bit-vector formulas that is based on MiniSat. As an alternative, CBMC has featured support for external SMT solvers since version 3.3. The solvers we recommend are (in no particular order) Boolector, MathSAT, Yices 2 and Z3. Note that these solvers need to be installed separately and have different licensing conditions.

Outline

Motivation : CWE Top-25

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere
[17]	65.5	CWE-732	Incorrect Permission Assignment for Critical Resource
[18]	64.6	CWE-676	Use of Potentially Dangerous Function
[19]	64.1	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
[20]	62.4	CWE-131	Incorrect Calculation of Buffer Size
[21]	61.5	CWE-307	Improper Restriction of Excessive Authentication Attempts
[22]	61.1	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
[23]	61.0	CWE-134	Uncontrolled Format String
[24]	60.3	CWE-190	Integer Overflow or Wraparound
[25]	59.9	CWE-759	Use of a One-Way Hash without a Salt

Format string : examples

```
printf ("The magic number is: %d\n", 1911);
printf ("The magic number is: \x25d\n", 23);
printf("Color %s, number1 %d, number2 %05d, hex %#x,
        float %5.2f, unsigned value %u.\n",
        "red", 123456, 89, 255, 3.14159, 250);
```

Format functions

- Convert C datatypes to string representation
- Allow to specify the format of the representation
- Process the resulting string : output to stderr, stdout,...
- fprint family : fprintf, printf, sprintf, snprintf, vfprintf, vprintf, vsprintf, vsnprintf
- Also setproctitle, syslog, err*, verr*, warn*, vwarn*

Format strings

Character	Description
%	Prints a literal % character (this type doesn't accept any flags, width, precision, length fields).
d , i	int as a signed decimal number. %d and %i are synonymous for output, but are different when used with scanf() for input (where using %i will interpret a number as hexadecimal if it's preceded by 0x , and octal if it's preceded by 0 .)
u	Print decimal unsigned int .
f , F	double in normal (fixed-point) notation. f and F only differs in how the strings for an infinite number or NaN are printed (inf , infinity and nan for f , INF , INFINITY and NAN for F).
e , E	double value in standard form ([-]d.ddd e [+ / -]ddd). An E conversion uses the letter E (rather than e) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00 . In Windows, the exponent contains three digits by default, e.g. 1.5e002 , but this can be altered by Microsoft-specific _set_output_format function.
g , G	double in either normal or exponential notation, whichever is more appropriate for its magnitude. g uses lower-case letters, G uses upper-case letters. This type differs slightly from fixed-point notation in that insignificant zeroes to the right of the decimal point are not included. Also, the decimal point is not included on whole numbers.
x , X	unsigned int as a hexadecimal number. x uses lower-case letters and X uses upper-case.
o	unsigned int in octal.
s	null-terminated string.
c	char (character).
p	void * (pointer to void) in an implementation-defined format.
a , A	double in hexadecimal notation, starting with 0x or 0X . a uses lower-case letters, A uses upper-case letters.[3][4] (C++11 iostreams have a hexfloat that works the same).
n	Print nothing, but writes the number of characters successfully written so far into an integer pointer parameter. Note: This can be utilized in Uncontrolled format string exploits.

Two equivalent (?) function calls

- Are the following instructions equivalent ?

```
printf (userinput);
printf ("%s", userinput);
```

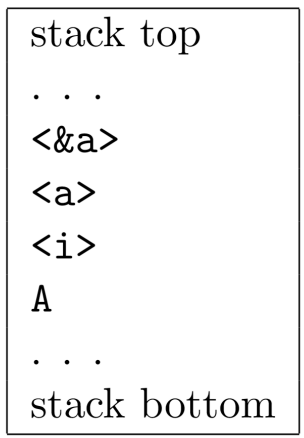
- What happens if userinput contains some format string special characters ?

Format functions (2)

- Format string should control format function behavior, specifies the type of parameters that should be printed
- Parameters are saved on the stack (pushed) either directly (by value), or indirectly (by reference)
- Stack constructed based on format function parameters (not the format string)
- Format function execution assumes that the stack is consistent with the format string

Format string : stack

```
printf ("Number %d has no address, number %d has:
        %08x\n", i, a, &a);
```



where:

A	address of the format string
i	value of the variable i
a	value of the variable a
&a	address of the variable i

Example and picture credit : Wenliang Du

Format string attacks

- Format strings mix data and "code" (special characters)
- Number of arguments is arbitrary
- Data types are not checked
- Potential impacts
  - Program crash
  - Memory leaks
  - Privilege escalation, with arbitrary code execution

Crashing the program

```
printf ("%s%s%s%s%s%s%s%s%s%s%s");
```

- "%s" displays memory from address supplied on the stack
- The stack also stores a lot of other data
- Chances are high to read from an illegal address

Reading the stack

```
printf ("%08x.%08x.%08x.%08x.%08x\n");
```

- Retrieves five parameters from the stack and displays them as 8-digit padded hexadecimal numbers
- In some cases you can retrieve entire stack memory
- Gives information on program flow and local variables
- May help finding correct offsets for successful exploitation

Reading memory at arbitrary place

```
printf ("\x10\x01\x48\x08_%08x.%08x.%08x.
         %08x.%08x|%s|");
```

- When processing "%s" fprint function will normally look at the content of the address stored on the stack
- Code includes just as many "%08x" needed so that printf will use the format string address for "%s"
- Format string starts with an arbitrary address location
- printf function will read memory from that address until reaching a NUL byte

Overwriting arbitrary memory locations

```
printf ("\x10\x01\x48\x08_%08x.%08x.%08x.
         %08x.%08x|%n|");
```

- When processing "%n" fprint function writes how many characters processed so far at address stored on stack
- Code includes just as many "%08x" needed so that printf will use the address in the format string
- Impact ? could change a privilege flag to non zero value

...

- Control value written using "%50d" instead of "%08x"

Protection/ detection

- When format string known at compile time :
  use your compiler !
  - gcc warnings -Wall,-Wformat, -Wno-format-extra-args, -Wformat-security, -Wformat-nonliteral, -Wformat=2
- If format string controlled by user : input validation

Outline

Summary

- Use safe language
- Use safe functions
- Check your inputs
- Listen to your compiler

References and Acknowledgements

- Smashing the stack for fun and profit, by Aleph One. `http://phrack.org/issues/49/14.html`
- Exploiting Format String Vulnerabilities, by scut / team teso `www.cis.syr.edu/~wedu/seed/Labs_12.04/Software/Format_String/files/formatstring-1.2.pdf`
- CWE webpages 120,121,122,134,190
- While preparing these slides I also used teaching material developed by Erik Tew at the University of Birmingham (kindly provided to me). Some of my slides are heavily inspired from his (but blame me for any errors !)