CS 88: Security and Privacy

05: Buffer Overflow and Format String Attacks

09-13-2022
Announcements

• Clicker mappings on edstem.
• Midterm dates set.
• Speak to me about accommodations now!
• Guest lecture next class

Reading Quiz
Heartbleed bug

[The project is ]"managed by four core European programmers, only one of whom counts it as his full-time job." The OpenSSL Foundation had a budget of less than $1 million in 2013.

- Wall Street Journal, Vox
Last Class

CS 31 Recap:
  • functions and the stack
  • assembly instructions
Today

• Software attacks
  • Stack Buffer Overflow
  • Format String Attacks
  • Heap overflow (shelphish)
Putting it all together...

Callee Code
1. push frame pointer
2. move stack pointer to frame pointer
3. increase stack pointer

Caller Code
1. save address of next instruction
2. push arguments

Callee’s frame.

Caller’s Frame Pointer
- Return Address
- First Argument to Callee
- Final Argument to Callee
- Caller’s local variables.

Caller’s frame.

Callee’s local variables.

Older stack frames.
...
Implementing a function call

```
main:
  ...
  subl $8, %esp
  movl $2, 4(%esp)
  movl $1, (%esp)
  call foo
  addl $8, %esp
  ...

foo:
  pushl %ebp
  movl %esp, %ebp
  subl $16, %esp
  movl $3, -4(%ebp)
  movl 8(%ebp), %eax
  addl $9, %eax
  leave
  ret
```
Arrays

void bar(char * in){
    char name[5];
    strcpy(name, in);
}

bar:
    pushl %ebp
    movl %esp, %ebp
    subl $5, %esp
    movl 8(%ebp), %eax
    movl %eax, 4(%esp)
    leal -5(%ebp), %eax
    movl %eax, (%esp)
    call strcpy
    leave
    ret
Data types / Endianness

x86 is a little-endian architecture

pushl %eax

%eax 0xfoo5ball

4 bytes

0x11 0xba 0x05 0x00

1 1 1 1

Higher Memory Addresses
Buffer Overflows
When is a program secure?

• Formal approach: When it does exactly what it should
  • not more
  • not less

• But how do we know what it is supposed to do?
Example 1

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

int main(int argc, char** argv) {
    char nice[] = "is nice."
    char name[8];
    gets(name);
    printf("%s %s\n", name, nice);
    return 0;
}
```
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv)
{
    char nice[] = "is nice.";
    char name[8];
    gets(name);
    printf("%s %s\n", name, nice);
    return 0;
}
```

What happens if we read a long name?

A. Nothing bad will happen
B. Something nonsensical will result
C. Something terrible will result
HOW THE HEARTBLEED BUG WORKS:

SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).

User Meg wants these 6 letters: POTATO. User Max wants pages about "irl games". Unlocking secure records with master key 5130985733433.

SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).

User Meg wants these 4 letters: BIRD. User Max wants pages about "os in car why". Note: Files for 4 is 975.381. 33.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 94 connections open. User Brendan uploaded the file.
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).

User: Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User: Karen wants to change account password to "password".

User: Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User: Karen wants to change account password to "password".
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaaa,0xbbbbbbbbb,argv[1]);
    return 0;
}
Buffer Overflow example

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

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

Load function arguments starting with the last argument
Buffer Overflow example

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xf005ball;
    char buf[4];
    strcpy(buf, str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa, 0xbbbbbbbbb, argv[1]);
    return 0;
}
```
Buffer Overflow example

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaaa,0xbbbbbbbbb,argv[1]);
    return 0;
}
```
Buffer Overflow example

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa,0xbbbbbbbbbb,argv[1]);
    return 0;
}
```
Buffer Overflow example

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf, str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa, 0xbbbbbbbbb, argv[1]);
    return 0;
}
```
Buffer Overflow example

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaaa,0xbbbbbbbbbb,argv[1]);
    return 0;
}
```
Buffer Overflow example

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf, str);
}

int main(int argc, char**argv) {
    func(0xaaaaaa, 0xbbbbbbb, argv[1]);
    return 0;
}
```
Buffer Overflow example

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaaa,0xbbbbbbbb,argv[1]);
    return 0;
}
```
Buffer Overflow example

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaa,0xbbbbbbbb,argv[1]);
    return 0;
}
```
Buffer Overflow example: If the first input is “AAAAAAAAAAAAAAAAA"

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaaa,0xbbbbbbbbb,argv[1]);
    return 0;
}
```
Buffer Overflow example: If the first input is "AAAAAAAAAAAAAAAAAA"

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaabbbbbb, 0xbabbbbbb, argv[1]);
    return 0;
}
```
Buffer Overflow example: If the first input is "AAAAAAAAAAAAAAAAAA"

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

void foo() { 0x08049b95
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball ;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaa,0xbbbbbbb,argv[1]);
    return 0;
}
```
Buffer Overflow example: If the first input is “AAAAAAAA\x95\x9b\x04\x08”

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

void foo() { 0x08049b95
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

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

<table>
<thead>
<tr>
<th>%esp</th>
<th>0x41141141</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebp</td>
<td>0x41141141</td>
</tr>
<tr>
<td>saved</td>
<td>0x08049b95</td>
</tr>
<tr>
<td>eip</td>
<td>0x41141141</td>
</tr>
<tr>
<td></td>
<td>0x41141141</td>
</tr>
</tbody>
</table>
Better Hijacking Control

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaaa,0xbbbbbbbb,argv[1]);
    return 0;
}
```
Better Hijacking Control

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

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

Jump to attacker supplied code where?
- put code in the string
- jump to start of the string
Shellcode

- Type of control flow hijack: taking control of the instruction pointer
- Small code fragment to which we transfer control
- Shellcode used to execute a shell
int main(void) {
    char* name[1];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
    return 0;
}

How do we transfer this to code?
Take the compiled assembly?
Payload is not always robust

Exact address of the shellcode start is not always easy to guess

Miss? Segfault
Fix? NOP Sled!
NOP Sled!

- NOP instruction: 0x90
- NOP sleds are used to pad out exploits
  - Composed of instruction sequences that don't affect proper execution of the attack
  - Classically the NOP instruction (0x90), but not restricted to that
- Why are the called sleds?
  - Execution *slides* down the NOPs into your payload
  - Overwritten return address can be less precise, so long as we land somewhere in the NOP sled

<table>
<thead>
<tr>
<th>badfile shellcode + NOP sled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x90 (NOP)</td>
</tr>
<tr>
<td>0x90 (NOP)</td>
</tr>
<tr>
<td>0x90 (NOP)</td>
</tr>
<tr>
<td>..</td>
</tr>
<tr>
<td>0x90 (NOP)</td>
</tr>
<tr>
<td>hijacked eip value</td>
</tr>
<tr>
<td>0x90 (NOP)</td>
</tr>
<tr>
<td>0x90 (NOP)</td>
</tr>
<tr>
<td>shellcode</td>
</tr>
</tbody>
</table>
Small Buffers

Buffer can be too small to hold exploit Code

Store exploit code in:
• an environmental variable
• or another buffer allocated on the stack
• redirect return address accordingly

badfile
shellcode +
NOP sled

hijacked eip value
0x90 (NOP)
0x90 (NOP)
long_buffer

environmental
variable
Putting it all together
Summary: Stack Code Injection

• Executable attack code is stored on stack, inside the buffer containing attacker’s string
  • Stack memory is supposed to contain only data, but...

• For the basic stack-smashing attack, overflow portion of the buffer must contain correct address of attack code in the RET position
  • The value in the RET position must point to the beginning of attack assembly code in the buffer
    • Otherwise application will crash with segmentation violation
  • Attacker must correctly guess in which stack position his buffer will be when the function is called
Draw out a stack diagram and build your very own shellcode attack

Information you are given:

- buffer to overflow:
  - char buffer[100]
  - &buffer[0] = 0xffffd88c
- $eip = 0xffffd8bc
- shellcode = 20 bytes
Buffer Overflow: Causes

• Typical memory exploit involves code injection
  • Put malicious code at a predictable location in memory, usually masquerading as data

• Trick vulnerable program into passing control to it
  • Overwrite saved EIP, function callback pointer, etc.
Buffer Overflow: can exploit…

A. pointer assignment & memory allocation, de-allocation
B. function pointers
C. calls to library routines
D. general purpose registers
E. format strings
Other Control Hijacking Opportunities: return-to-libc attack

(1) Change the return address to point to the attack code. After the function returns, control is transferred to the attack code.

(2) ... or return-to-libc: use existing instructions in the code segment such as system(), exec(), etc. as the attack code.

① set stack pointers to return to a dangerous library function

② /bin/sh
(1) Change a function pointer to point to the attack code

(2) Any memory, on or off the stack, can be modified by a statement that stores a compromised value into the compromised pointer. 
```c
strcpy(buf, str);  *ptr = buf[0];
```
Other Control Hijacking Opportunities: Frame Pointer

Change the caller’s saved frame pointer to point to attacker-controlled memory.
Caller’s return address will be read from this memory.
Some Unsafe C lib Functions

`strcpy(char *dest, const char *src)`
`strcat(char *dest, const char *src)`
`gets(char *s)`
`scanf(const char *format, ...)`
`printf(const char *format, ...)`
Avoid strcpy, ...

• We have seen that `strcpy` is unsafe
  • `strcpy(buf, str)` simply copies memory contents into `buf` starting from `*str` until “\0” is encountered, ignoring the size of `buf`
  • Avoid `strcpy()`, `strcat()`, `gets()`, etc.
  • Use `strncpy()`, `strncat()`, instead

• Even these are not perfect... (e.g., no null termination)

• Always a good idea to do your own validation when obtaining input from untrusted source

• Still need to be careful when copying multiple inputs into a buffer
Cause of vulnerability: No Range Checking

- `strcpy` does not check input size
  - `strcpy(buf, str)` simply copies memory contents into `buf` starting from `*str` until `"\0"` is encountered, ignoring the size of area allocated to `buf`
Does Range Checking Help?

\texttt{strncpy}(\texttt{char *dest, const char *src, size_t n})

- copy no more than n characters from source to destination
- contingent on? the right value of n!

• Potential overflow in \texttt{htpasswd.c} (Apache 1.3):

  \begin{verbatim}
  ... \texttt{strcpy\(\)(record, user)};\
  \texttt{strcat\(\)(record,"\":");}\
  \texttt{strcat\(\)(record, cpw);} ...\
  \end{verbatim}

  Copies username ("user") into buffer ("record"), then appends ":" and hashed password ("cpw")

• Published fix:

  \begin{verbatim}
  \texttt{strncpy\(\)(record, user, MAX\_STRING\_LEN\-1);}\
  \texttt{strcat\(\)(record,"\":");}\
  \texttt{strncat\(\)(record, cpw, MAX\_STRING\_LEN\-1);} ...\
  \end{verbatim}

A. The fix ensures that there are no vulnerabilities
B. The vulnerabilities are still present.
Integer overflows

A) This code is free from integer overflow vulnerabilities.
B) Integer vulnerabilities still exist.

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

int main(int argc, char *argv[])
{
    unsigned short s;
    int i;
    char buf[80];

    if(argc < 3){
        return -1;
    }

    i = atoi(argv[1]);
    s = i;

    if(s >= 80) {
        /* [w1] */
        printf("Oh no you don't!\n");
        return -1;
    }

    printf("s = %d\n", s);

    memcpy(buf, argv[2], i);
    buf[i] = '\0';
    printf("%s\n", buf);

    return 0;
}
```
Width Overflows

```c
uint32_t x = 0x10000;
uint16_t y = 1;
uint16_t z = x + y; // z = ?
```

- Width overflows occur when assignments are made to variables that can't store the result.
- Integer promotion
  - Computation involving two variables `x, y` where `width(x) > width(y)`
  - `y` is promoted such that `width(x) = width(y)`
Sign Overflows

int f(char* buf, int len) {
    char dst_buf[64];
    if (len > 64)
        return 1;
    memcpy(dst_buf, buf, len);
    return 0;
}

• Sign overflows occur when an unsigned variable is treated as signed, or vice-versa
  • Can occur when mixing signed and unsigned variables in an expression
  • Or, wraparound when performing arithmetic
Broward Vote-Counting Blunder Changes Amendment Result

POSTED: 1:34 pm EST November 4, 2004

BROWARD COUNTY, Fla. -- The Broward County Elections Department has egg on its face today after a computer glitch misreported a key amendment race, according to WPLG-TV in Miami.

Amendment 4, which would allow Miami-Dade and Broward counties to hold a future election to decide if slot machines should be allowed at racetracks, was thought to be tied. But now that a computer glitch for machines counting absentee ballots has been exposed, it turns out the amendment passed.

"The software is not geared to count more than 32,000 votes in a precinct. So what happens when it gets to 32,000 is the software starts counting backward," said Broward County Mayor Ilene Lieberman.

Broward County Mayor Ilene Lieberman says voting counting error is an "embarrassing mistake."

That means that Amendment 4 passed in Broward County by more than 240,000 votes rather than the 166,000-vote margin reported Wednesday night. That increase changes the overall statewide results in what had been a neck-and-neck race, one for which recounts had been going on today. But with news of Broward’s error, it's clear amendment 4 passed.
Heartbleed vulnerability

```c
struct {
    HeartbeatMessageType type;
    #highlight
    uint16 payload_length;
    uchar payload [HeartbeatMessage.payload_length];
    uchar padding[padding_length];
} HeartbeatMessage;
```
Heartbleed vulnerability

*Figure 1. Attack activity related to the Heartbleed vulnerability, as noted for IBM Managed Security Services customers, in April 2014*
Off-By-One Overflow

Home-brewed range-checking string copy

```c
void notSoSafeCopy(char *input) {
    char buffer[512]; int i;
    for (i=0; i<=512; i++)
        buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
    if (argc==2)
        notSoSafeCopy(argv[1]);
}
```

This will copy 513 characters into buffer. Oops!

1-byte overflow: can’t change RET, but can change saved pointer to previous stack frame
What’s wrong with this code?

```c
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }
    memcpy(buf, p, len);
}

void *memcpy(void *dest, const void *src, size_t n);
    typedef unsigned int size_t;
```

A. Nothing
B. Buffer overflow
C. Integer overflow
D. Race Condition
Other overflow targets

• Format strings in C
• Heap management structures used by malloc
Format String Vulnerabilities
Variable arguments in C

In C, we can define a function with a variable number of arguments

```c
void printf(const char* format,....)
```

Usage:

```c
printf(“hello world”);
printf(“length of %s = %d \n”, str, str.length());
```

format specification encoded by special % characters
fun with format strings

printf("you scored %d\n", score);

stack base pointer
return address
arg1: 0x08048464
arg2: score

\0 \n d
% d e
r o c s
u o y
Implementation of printf

- Special functions `va_start`, `va_arg`, `va_end` compute arguments at run-time

```c
void printf(const char* format, ...) {
    int i; char c; char* s; double d;
    va_list ap; /* declare an “argument pointer” to a variable arg list */
    va_start(ap, format); /* initialize arg pointer using last known arg */

    for (char* p = format; *p != '\0'; p++) {
        if (*p == '%') {
            switch (*++p) {
                case 'd':
                    i = va_arg(ap, int); break;
                case 's':
                    s = va_arg(ap, char*); break;
                case 'c':
                    c = va_arg(ap, char); break;
            }
        ... /* etc. for each % specification */
    }
    ... /* etc. for each % specification */

    va_end(ap); /* restore any special stack manipulations */
}
```
fun with format strings

printf(“a %s costs $%d
”, item, price);
Closer look at the stack

```c
printf("Numbers: %d, %d", 5, 6);
```

Internal stack pointer starts here

```
Addr 0xFF...F
```

Local variables

```
Saved FP  ret/IP  &str  5  6  Caller's frame
```

```
printf("Numbers: %d, %d");
```

Internal stack pointer starts here

```
Addr 0xFF...F
```

Local variables

```
...  Saved FP  ret/IP  &str  Caller's frame
```

Args
Sloppy use of printf

```c
void main(int argc, char* argv[])
{
    printf( argv[1] );
}
```

Attacker controls format string gives all sorts of control:
- Print stack contents
- Print arbitrary memory
- Write to arbitrary memory
Format specification encoded by special % characters

**Format Specifiers**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Passed as</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>decimal (int)</td>
<td>value</td>
</tr>
<tr>
<td>%u</td>
<td>unsigned decimal (unsigned int)</td>
<td>value</td>
</tr>
<tr>
<td>%x</td>
<td>hexadecimal (unsigned int)</td>
<td>value</td>
</tr>
<tr>
<td>%s</td>
<td>string ((const) (unsigned) char *)</td>
<td>reference</td>
</tr>
<tr>
<td>%n</td>
<td>number of bytes written so far, (* int)</td>
<td>reference</td>
</tr>
</tbody>
</table>
The %n format specifier

- %n format symbol tells printf to write the number of characters that have been printed
  - Argument of printf is interpreted as a destination address

- printf ("overflow this!%n", &myVar);
  - Writes 14 into myVar.
The %n format specifier

• %n format symbol tells printf to write the number of characters that have been printed
  • Argument of printf is interpreted as a destination address

• printf ("overflow this!%n", &myVar);
  • Writes 14 into myVar.

• What if printf does not have an argument?
  • char buf[16] = "Overflow this!%n";
  • printf(buf);

A. Store the value 14 in buf
B. Store the value 14 on the stack (specify where)
C. Replace the string Overflow with 14
D. Something else
Viewing the stack

We can show some parts of the stack memory by using a format string like this:

C code

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

Output

```
40012980.080628c4.bff7a4.00000005.08059c04
```

instruct printf:

- retrieve 5 parameters
- display them as 8-digit padded hexademical numbers
fun with printf: what’s the output of the following statements?

```c
printf("100% dive into C!");
printf("100% samy worm");
printf("%d %d %d %d");
printf("%d %s");
printf("100% not another segfault!");
```
Heap based buffer overflow

- Heap stores “chunks” of memory using inked lists
- when malloc is called:
  - stores “meta data” about the chunk right above the newly allocated block
- metadata can be exploited to corrupt memory

```
struct Node{
    struct Node *next;
    struct Node *pre;
};

// remove Node p from the linked list
q = p->pre;
q->next = p->next;
```
Heap Overflow Exploit Techniques

Overwrite next pointer in linked list effectively the same as overwriting the return address on the stack when the malloc function is next involved: control flow is hijacked to point to the attackers code.

Heap Buffer Overflow
- a buffer on the heap is not checked
- attacker writes beyond the end of the allocated chunk and corrupts the pointer.

Lots of different variations:
- use after free
- double free
- unlink exploit
- shrinking free chunks..
- house of spirit...
<table>
<thead>
<tr>
<th>Implementation</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptmalloc2</td>
<td>Linux, HURD (glibc)</td>
</tr>
<tr>
<td>SysV AT&amp;T</td>
<td>IRIX, SunOS</td>
</tr>
<tr>
<td>Yorktown</td>
<td>AIX</td>
</tr>
<tr>
<td>RtIHeap</td>
<td>Windows</td>
</tr>
<tr>
<td>tcmalloc</td>
<td>Google and others</td>
</tr>
<tr>
<td>jemalloc</td>
<td>FreeBSD, NetBSD, Mozilla</td>
</tr>
<tr>
<td>phkmalloco</td>
<td>*BSD</td>
</tr>
</tbody>
</table>
ptmalloc

- Extremely popular malloc (default in glibc)
- Stores memory management metadata inline with user data
  - Stored as small chunks before and after user chunks
- Aggressive optimizations
  - Maintains lists of free chunks binned by size
  - Merges consecutive free chunks to avoid fragmentation
Use after free

Consider the sample code:

```c
char *a = malloc(20); // 0xe4b010
char *b = malloc(20); // 0xe4b030
char *c = malloc(20); // 0xe4b050
char *d = malloc(20); // 0xe4b070

free(a);
free(b);
free(c);
free(d);

a = malloc(20); // 0xe4b070
b = malloc(20); // 0xe4b050
c = malloc(20); // 0xe4b030
d = malloc(20); // 0xe4b010
```

The state of the particular fastbin progresses as:

1. 'a' freed.
   ```
   head -> a -> tail
   ```
2. 'b' freed.
   ```
   head -> b -> a -> tail
   ```
3. 'c' freed.
   ```
   head -> c -> b -> a -> tail
   ```
4. 'd' freed.
   ```
   head -> d -> c -> b -> a -> tail
   ```
5. 'malloc' request.
   ```
   head -> c -> b -> a -> tail ['d' is returned]
   ```
6. 'malloc' request.
   ```
   head -> b -> a -> tail ['c' is returned]
   ```
7. 'malloc' request.
   ```
   head -> a -> tail ['b' is returned]
   ```
8. 'malloc' request.
   ```
   head -> tail ['a' is returned]
   ```

Source: https://heap-exploitation.dhavalkapil.com/attacks/
Double free

Consider this sample code:

```c
a = malloc(10); // 0xa04010
b = malloc(10); // 0xa04030
c = malloc(10); // 0xa04050

free(a);
free(b); // To bypass "double free or corruption (fasttop)
free(a); // Double Free !
```

d = malloc(10); // 0xa04010
e = malloc(10); // 0xa04030
f = malloc(10); // 0xa04010 - Same as 'd'!

The state of the particular fastbin progresses as:

1. 'a' freed.
   - head -> a -> tail
2. 'b' freed.
   - head -> b -> a -> tail
3. 'a' freed again.
   - head -> a -> b -> a -> tail
4. 'malloc' request for 'd'.
   - head -> b -> a -> tail ['a' is returned]
5. 'malloc' request for 'e'.
   - head -> a -> tail ['b' is returned]
6. 'malloc' request for 'f'.
   - head -> tail ['a' is returned]

Source: https://heap-exploitation.dhavalkapil.com/attacks/