Recap

Stack overflows corrupt memory on the stack allowing to overwrite/control
  ▪ Return addresses (control-flow hijacking)
  ▪ Other data saved in the stack

Global and heap buffer overflows corrupt neighboring memory allowing to overwrite/control
  ▪ Other data saved in the stack

Controlling the return address can lead to code injection and arbitrary code execution

Controlling program data can lead to unexpected/undesired behavior
More Attacks

Heap overflows as arbitrary writes

Format string exploits
More Attacks

Heap overflows as arbitrary writes

Format string exploits
Understanding the Heap

The layout of buffers in memory depends on the implementation of the allocator (i.e., malloc).

```c
char *userinput = malloc(20);
char *outputfile = malloc(20);
```
malloc() Implementations

dlmalloc – General purpose allocator
ptmalloc2 – glibc
jemalloc – FreeBSD and Firefox
tcmalloc – Google
libumem – Solaris
...

Heap memory is obtained from the kernel using the brk() or mmap() system calls
  - Provides plenty of “raw” space

The allocator splits memory into **arenas**
  - Each thread gets its own arena
  - Each arena has its own metadata

Memory within the arena is split into **chunks** and given to program through various allocation functions (e.g., malloc())
  - Chunks are organized in bins, usually through double linked-lists

https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/
Buffer/Metadata Interleaving

Memory management metadata
Corrupted Metadata

Use of the corrupted meta data and may lead to an arbitrary write, corrupting a code pointer or security critical data.
Heap Arena Structure

No two free chunks can be adjacent.
Heap Arena Structure

No two free chunks can be adjacent.

Adjacent free chunks are merged together
If previous chunk is free, this field contains size of previous chunk, else user data.

This chunk size

N
M
P

User data

Previous size field contains user data since its previous chunk is allocated.

Bitmap

- P - This bit is set when previous chunk is allocated
- M - This bit is set when chunk is mmap’d
- N - This bit is set when this chunk belongs to a thread arena.
Free Chunk

- **chunk**: User data of previous chunk since previous chunk is allocated.
- **mem**: `fd` - Points to next chunk in the binlist.
- **next chunk**: `bk` - Points to previous chunk in the binlist.
- **Unused Space**: Previous size field contains its previous chunk’s size since previous chunk is free.
<table>
<thead>
<tr>
<th>This chunk size</th>
<th>N</th>
<th>M</th>
<th>P</th>
</tr>
</thead>
</table>

If previous chunk is free, this field contains size of previous chunk, else user data.

User data

Corrupted metadata

Previous size field contains user data since its previous chunk is allocated.

User data

Next chunk

Mem

Mem

Chunk
Linked-list Manipulation to Arbitrary Write

Original list, with a pointer to a node to be removed:

L: item 1 ← item 2 → item 3 → ... → item N

n:

Step 1: Change the prev field of the node to the right of node n:

L: item 1 ← item 2 → item 3 → ... → item N

n:

Step 2: Change the next field of the node to the left of node n (n is now removed from the list):

L: item 1 ← item 2 → item 3 → ... → item N

n:

Corrupted pointers attacker controlled next and prev pointers due to the overwritten n

Remove n

n->next->prev = n->prev;

n->prev->next = n->next;
Linked-list Manipulation to Arbitrary Write

Original list, with a pointer to a node to be removed:

\[
L: \quad \rightarrow \text{item 1} \rightarrow \text{item 2} \rightarrow \text{item 3} \rightarrow \cdots \rightarrow \text{item N} \\
\quad \rightarrow \text{n: } \quad \rightarrow
\]

Step 1: Change the prev field of the node to the right of node n:

\[
L: \quad \rightarrow \text{item 1} \rightarrow \text{item 2} \rightarrow \text{item 3} \rightarrow \cdots \rightarrow \text{item N} \\
\quad \rightarrow \text{n: } \quad \rightarrow
\]

\[
*(n->\text{next} + \text{prev_offset}) = n->\text{next} \\
n->\text{next}->\text{prev} = n->\text{prev};
\]

Step 2: Change the next field of the node to the left of node n (n is now removed from the list):

\[
L: \quad \rightarrow \text{item 1} \rightarrow \text{item 2} \rightarrow \text{item 3} \rightarrow \cdots \rightarrow \text{item N} \\
\quad \rightarrow \text{n: } \quad \rightarrow
\]

\[
*(n->\text{prev} + \text{next_offset}) = n->\text{next} \\
n->\text{prev}->\text{next} = n->\text{next};
\]
Example 1

```c
int main(int argc, char **argv)
{
    int i;
    char *buf1;

    buf1 = malloc(64);
    for (i = 0; i < 200; i++)
        buf1[i] = 'A';
    return 0;
}
```

```c
int main(int argc, char **argv)
{
    int i;
    char *buf1;

    buf1 = malloc(64);
    for (i = 0; i < 200; i++)
        buf1[i] = 'A';
    free(buf1);
    return 0;
}
```
```c
int main(int argc, char **argv)
{
    int i;
    char *buf1, *buf2;

    buf1 = malloc(64);
    buf2 = malloc(64);
    for (i = 0; i < 200; i++)
        buf2[i] = buf1[i] = 'A';
    free(buf2);
    free(buf1);
    return 0;
}
```
Example 2

```c
int main(int argc, char **argv)
{
    int i;
    char *buf1, *buf2;

    buf1 = malloc(64);
    buf2 = malloc(64);
    for (i = 0; i < 200; i++)
        buf2[i] = buf1[i] = 'A';
    free(buf2);
    free(buf1);
    return 0;
}
```

Program received signal SIGSEGV, Segmentation fault.
_int_free (av=0x7fffffff7dd6620 <main_arena>, p=0x601050, have_lock=0)
at malloc.c:3966
int main(int argc, char **argv)
{
    int i;
    char *buf1, *buf2, *buf15;

    buf1 = malloc(64);
    buf15 = malloc(200);
    buf2 = malloc(64);
    for (i = 0; i < 200; i++)
        buf15[i] = buf2[i] = buf1[i] = 'A';
    free(buf2);
    free(buf1);
    return 0;
}
Double-Free Bugs

Freeing the same buffer twice can also lead to metadata corruption

- May be harder to exploit

```c
int main(int argc, char **argv)
{
    int i;
    char *buf1, *buf2;

    buf1 = malloc(200);
    buf2 = malloc(200);
    for (i = 0; i < 200; i++)
        buf2[i] = buf1[i] = 'A';
    free(buf2);
    free(buf2);
    return 0;
}
```
Heap Overflows In Practice

Exploiting the allocator depends on
- The allocator’s implementation
- The sequence of allocator calls in the program

The attacker may need to “guide” the program to perform a long sequence of allocations and deallocations to align the objects in the heap.
More Attacks

Heap overflows as arbitrary writes

Format string exploits
Format String Bugs

Occurs when untrusted input is used as format string

Exploits how variadic functions and the printf-family of functions specifically work

```
int printf(const char * restrict format, ...);
```
Argument Types and Number Based on Format String

printf("%ld %h %c %g %s", long_integer, short, character, double, string);

Arguments are pushed to the stack!

printf reads stack arguments based on the format string
Not Enough Arguments

```c
printf("%ld %h %c %g %s");
```

What happens when there is a mismatch between format string and actual arguments?
Not Enough Arguments

```c
printf(“%ld %h %c %g %s”);
```

What happens when there is a mismatch between format string and actual arguments?

Memory (stack) data are leaked

- High addresses
  - **RSP**
  - `long int`

---

Stevens Institute of Technology
Not Enough Arguments

printf(“%ld %h %c %g %s”);

What happens when there is a mismatch between format string and actual arguments?

Memory (stack) data are leaked
Not Enough Arguments

`printf("%ld %h %c %g %s");`

What happens when there is a mismatch between format string and actual arguments?

Memory (stack) data are leaked

High addresses

- char
- short
- long int

RSP
Direct Parameter Access

“%3$x” → Access the 3\textsuperscript{rd} argument

High addresses

RSP

tack
Corrupting Memory Using printf

%\n can be used to store the number of written characters into an integer pointer

int n;
long li = 100;
printf(“%ld\n%\n”, li, &n);
Corrupting Memory Using printf

\%n can be used to store the number of written characters into an integer pointer

```c
int n;
long li = 100;
printf("%ld\n%n", li, &n);

n = 4
```
Corrupting Memory Using printf

printf("%ld$3n", li);
More `printf()`

Length modifier (+ zero padding)

```c
long li = 23;
printf("%0128ld\n", li);
```

```
000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000
00000000000023
```

It is easy to write a large number of characters!
printf As An Arbitrary Write

printf("%0128ld%$3n", li);

High addresses

128  \texttt{tack}  li  RSP