C4 Integer Overflows & Targeted Overwrites

1. Security Issues with Integers

1.1 Background on Integers

Natural Numbers

- The natural numbers (counting numbers) $\mathbb{N}$ are defined by the Peano postulates:
  - 1 is a member of the set $\mathbb{N}$
  - If n is a member of $\mathbb{N}$, then the successor $s(n)$ belongs to $\mathbb{N}$
  - 1 is not the successor of any element in $\mathbb{N}$
  - If $s(n) = s(m)$, then $n = m$
  - If $m \in \mathbb{N}$ is not 1, then there exists a $n \in \mathbb{N}$ such that $s(n) = m$
  - A subset of $\mathbb{N}$ which contains 1, and which contains $n + 1$ whenever it contains n, must equal $\mathbb{N}$
- You can write $n + 1$ instead of $s(n)$

- $a, b \in \mathbb{N}$, if $b = 1$, then $a + b = s(a)$, else take $c \in \mathbb{N}$ so that $s(c) = b$ and define $a + b = s(a + c)$

1.2 Integer Overflows

- Unsigned 8-bit integers

  $255 + 1 = 0$ $\quad$ $16 \times 17 = 16$ $\quad$ $0 - 1 = 255$
- Unsigned 64-bit integers

  $2^{63} + 2^{63} = 0$
- Signed 8-bit integers

  $127 + 1 = -128$ $\quad$ $-128 \div -1 = -128$

- Can lead to buffer overflows (bof)
  - OS checks string lengths to defend against bof

- "safe" signed addition prevents addition of arguments with the same sign

1.3 Integer Mismatches

- Declare all integers as unsigned integers unless negative numbers are needed (compilers will issue warning if signed-unsigned mismatch occurs)
- **Truncation:** input UID as signed integer, check value $\neq 0$, truncate to unsigned short integer $0\text{x}10000 \rightarrow 0\text{x}0000 \text{ (root!)}$
- Integer overflows happen when behaviours of abstraction and implementation diverge
  - Place checks in code or use libraries that detect such situation and flag error so that caller can handle these error messages correctly

2. Format String Attacks

- Overwriting arbitrary pointer with an arbitrary value
- Bof overwrites all position between buffer and target which may crash a program before returning
- In format string attacks, `printf()` does the attacker's job
- In double-free attacks, `malloc()` does the attacker's job

2.1 printf()

`printf(char *, …);`	creates a formatted string and writes it to standard out I/O stream
`fprintf(FILE *, char *, …);`	creates a formatted string and writes it to a libc FILE I/O stream
`sprintf(char *, char *, …);`	creates a formatted string and writes it to a location in memory
`snprintf(char *, size_t, char *, …);`	creates a formatted string and writes it to a location in memory, with a maximum string size

`%i`, `%d`	`int`, `short`, `char`	Integer value of argument in decimal notation
`%u`	`unsigned int`, `short`, `char`	Value of argument as unsigned integer in decimal notation
`%x`	`unsigned int`, `short`, `char`	Value of argument as unsigned integer in hexadecimal notation
`%s`	`char *`, `char[]`	Character string pointed to by the argument
`%p`	`(void *)`	Value of the pointer in hexadecimal notation
`%n`	`(int *)`	Number of bytes output so far, stored in an argument that is a pointer to an integer

.precision: here 100 digits for rendering **x**

```
char buf[10];
int ctr, x = 0;
snprintf(buf, sizeof buf,
                 "%.100d%n", x, &ctr);
printf("counter: %d\n", ctr);
```

- Counter is incremented to 100, although only 10 characters are actually written to the buffer

- `snprintf()` saves number of characters printed out to address allocated to ctr
- `printf("%.100d%n", x, &ctr)` works the same way, although there is no maximum string size

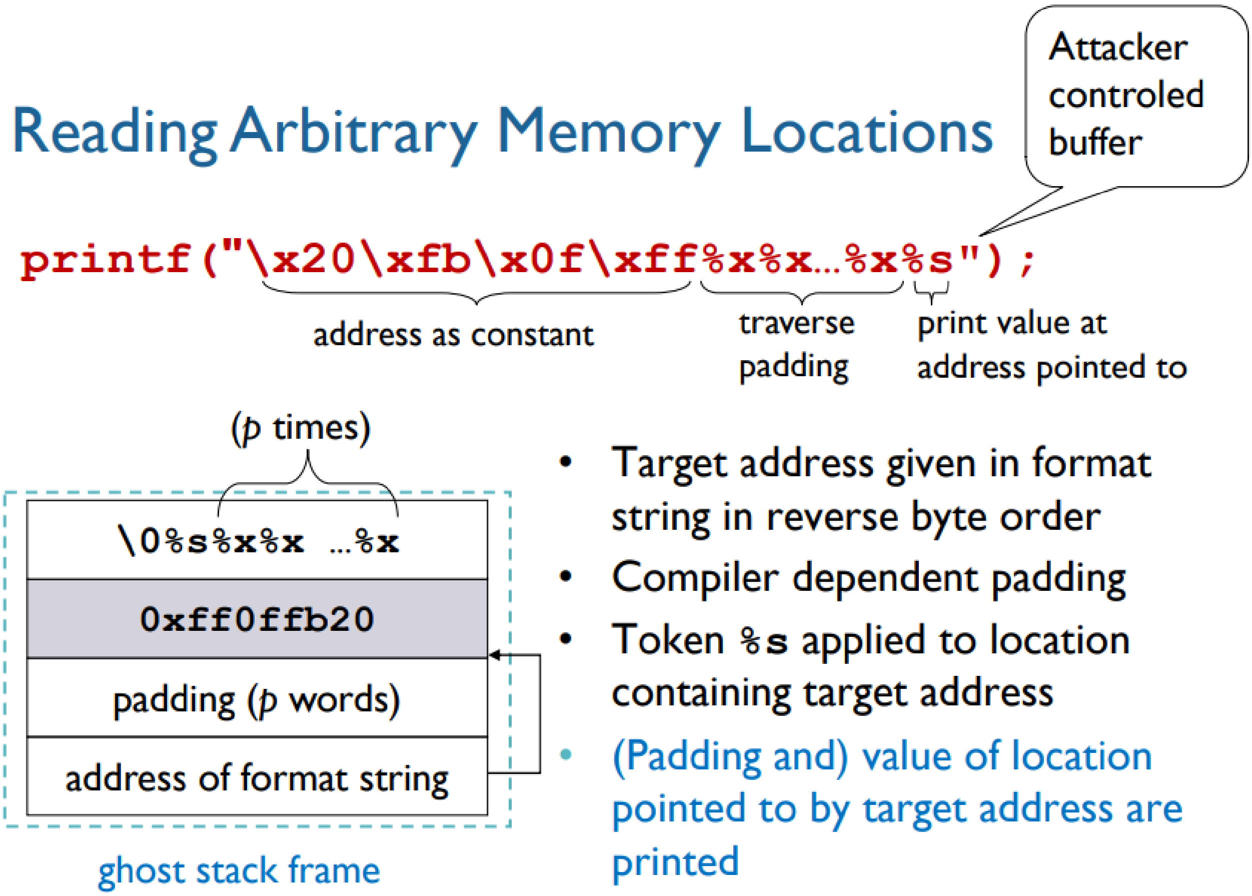
Execution

- Format string & arguments provided to printf() allocated on the stack

- Stack frame for printf() contains arguments passed to the function
    - **Assume** address of format string passed in as argument
- No. of arguments calculated by counting **format tokens (e.g. %c, %d)** in the format string
    - If no. of tokens > no. of arguments, memory positions next in stack frame will be printed (in 64 bit linux, parameters passing is done in this order: RDI, RSI, RDX, RCX, R8, R9, remaining from the stack)

Reading Arbitrary Memory Locations

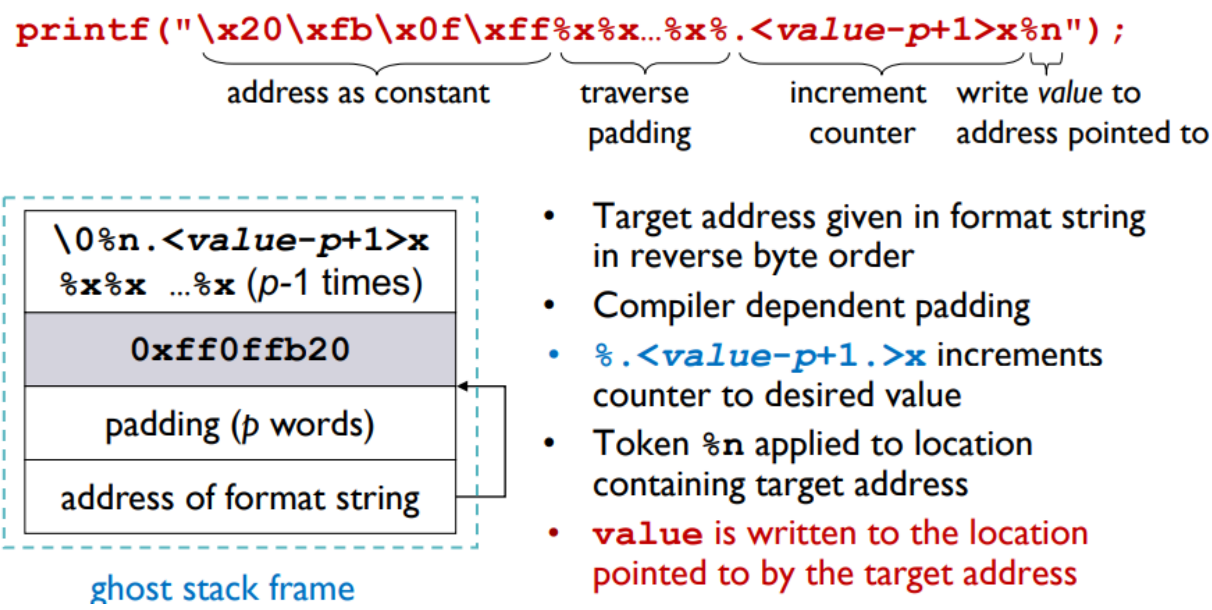
- Identify format string function (e.g. `printf()`) where you get to specify format string
- When `printf()` is called without format tokens, an attack can pass its own format tokens to create a *ghost* stack frame
- To read from a chosen memory address, put the address as a constant in the format string
- Construct format string so that this constant becomes argument for a %s (char array) format token in the ghost frame



Writing to Arbitrary Memory Locations

- Include format tokens to traverse memory locations from location the first argument is expected to be at to the address of the format string

- Use `.precision` to set counter value of your choice



(similar vid: https://www.youtube.com/watch?v=CyazDp-Kkr0)

3. Heap Memory Allocation

3.1 Malloc() and Free()

Malloc()

- `void * malloc(size)` returns pointer to newly allocated block of `size` bytes
  - contents of block are not initialised
  - returns null pointer if call fails

Free()

- `void free (void *ptr)`
  - `ptr` must come from previously called `malloc, calloc, realloc`
  - no operation performed if `ptr` is null
  - if called again, behaviour is undefined
  - **not atomic (not done in one instruction call)**

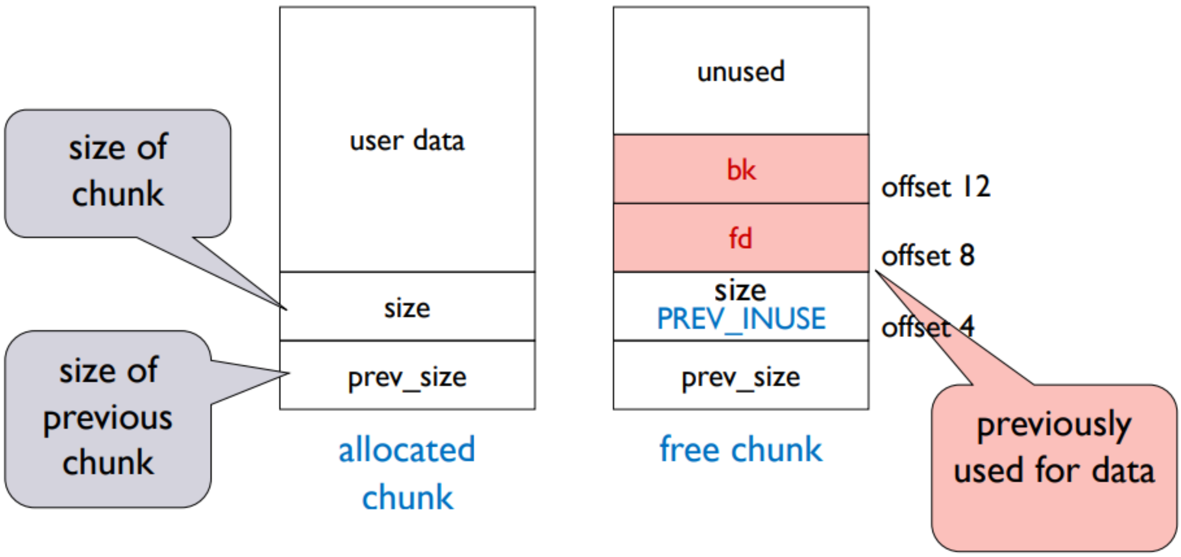
3.2 Doubly Linked Lists for Memory Management

Memory Organisation

- Memory is divided into chunks which contain user data and control data
  - Chunks allocated by malloc contain boundary tags
  - Free chunks are placed in bins (doubly linked list)
    - Tags of a free chunk contain a forward (fd) and backward (bk) pointer to its neighbours in the bin (can have allocated chunks in between)

Allocated and Free Chunks

```
struct chunk {
  int prev_size;
  int size;
  /* used of if free */
  struct chunk *fd;
  struct chunk *bk;
}
```



  - Allocated chunks do not have pointers to neighbouring chunks

- Values for `size` is always given in multiples of 8 with the 3 LSBs used as control flags (`PREV_INUSE (0x1), IS_MAPPED (0x2)`, some libraries use the 3rd bit)
  - `[size][??][mapped][prev_chunk_inuse]`
  - When a chunk is freed it is combined into a single chunk with its neighbours
  - `PREV_INUSE` is used to check if the previous chunk is in use, if not, combine both chunks

Two Views on Memory

- "Topological" (physical) view: location of chunks in memory
  - Chunks ordered by their addresses
  - We can talk about the "chunk above" and the "chunk below"
  - This view matters when chunks are coalesced
- "Logical" view: location of chunks in a bin
  - Chunks ordered by their position in a list
  - We can talk about "previous chunk" and "next chunk"
  - This view matters when chunks are allocated and freed

Bin Management

- Doubly-linked lists of free chunks **ordered by increasing size** to facilitate fast smallest-first search

- `unlink()` is used to allocate a buffer during `malloc()`
- `frontlink()` is used to insert a freed chunk to the right position (by size)

Frontlink()

- **Simplified version**

```
#define frontlink(A, P, S, IDX, BK, FD)  ...
[1] FD = start_of_bin(IDX);
[2] while ( FD != BK && S < chunksize(FD) )
     {
[3]        FD = FD->fd;
     }
[4]  BK = FD->bk;
[5]  P->bk = BK;
[6]  P->fd = FD;
[7]  FD->bk = BK->fd = P;
}
```

  - Store chunk of size `S`, pointed to by `P` at appropriate position in the double linked list of bin with index `IDX`

- **Macro**

  [1] FD initialized with a pointer to the start of the list of the given bin

  [2] Loop searches the double linked list to find first chunk not larger than P or the end of the list by following consecutive forward pointers (line 3)

  [4] Follow back pointer BK to previous element in list

  [5]+[6] Set backward and forward pointers for chunk P

  [7] Update backward pointer of next chunk & forward pointer of previous chunk to address of chunk P (field fd at 8 byte offset within a boundary tag)

Unlink()

- **Simplified version**

```
#define unlink(P, BK, FD)
{
[1] FD = P->fd;
[2] BK = P->bk;
[3] FD->bk = BK;
[4] BK->fd = FD;
}
```

- **Macro**
  - [1] [2] Save pointers in chunk P to FD and BK
  - [3] Update backward pointer of next chunk in the list: address located at FD plus 12 bytes (offset of *bk* field in boundary tag) overwritten with value stored in BK
  - [4] Update forward pointer of the previous chunk

4. Double-free vulnerabilities

4.1 Double Free Attack

- Call `free(A)` with forward consolidation after allocation of memory chunk `A` to create a larger chunk
- Allocate chunk `B`, hoping to get space just freed
- Copy ghost chunk into `B` at the location of `A` and a free ghost chunk adjacent to the chunk at `A`
  - ghost chunk is written into B and `free(A)` is called
- Calling `free(A)` again which coalesces the two ghost chunks and will try to remove the free ghost chunk from its bin

Removing ghost chunk

- Attacker writes fake forward and backward pointers into first 8 bytes of free ghost chunk
  - fd: target address to be overwritten, minus 12
  - bk: value to be written to the target address (fd + 12)
- Unlinking free ghost chunk uses fake pointers fd and bk
- Deallocation of heap is not atomic
  - Second call of `free(A)` only has the desired effect if the pointer to A had not been set to null when A was freed the first time
- **Issue caused by function releasing memory but not setting pointer to null**

4.2 Defences

- Stay true to abstraction: do not apply unlink to a chunk that is not part of a double link list
- Protect boundary tags with canaries
  - When chunk is allocated, initialise canary
  - When chunk is freed, recalculate checksum and compare with canary
- Randomise memory allocation so attacker cannot predict next allocated chunk
- Split user data from control data