macOS

How to gain root with CVE-2018-4193 in < 10s

Date 16th of February 2019

At OffensiveCon 2019

By Eloi Benoist-Vanderbeken

Whoami

- **Eloi Benoist-Vanderbeken**
- **@elvanderb on twitter**

- **Working for Synacktiv:**
  - Offensive security company
  - 50 ninjas
  - 3 poles: pentest, reverse engineering, development

- **Reverse engineering team coordinator:**
  - 21 reversers
  - Focus on low level dev, reverse, vulnerability research/exploitation
  - If there is software in it, we can own it :)
  - We are hiring!

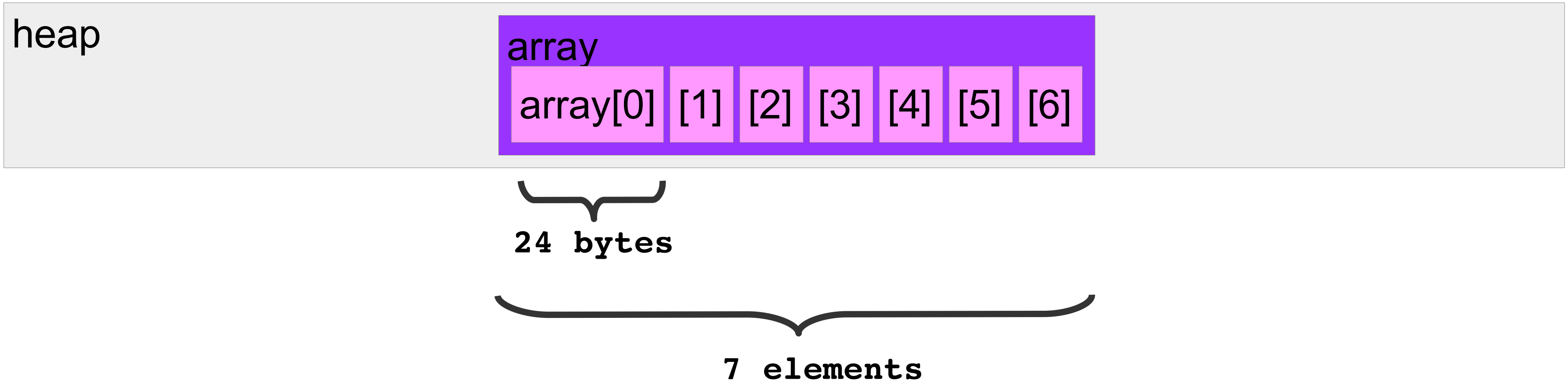
Introduction

CVE-2018-4193

- **Vulnerability in WindowServer**
  - Userland macOS root service
  - “WindowServer is a system daemon that provides various UI services such as window management, content compositing, and event routing”
  - Fixed in macOS 10.13.5
- **Discovered by ret2 Systems**
  - And at least 2 other pwn2own 2018 teams
  - https://twitter.com/_niklasb/status/1004342074114760704
- **Used in a pwn2own 2018 chain**
  - ~90s to spawn a root shell
  - Unfortunately pwn2own only allows 3 attempts and their exploit worked the 4th time
- **Full chain described in an excellent blog post series**
  - https://blog.ret2.io/2018/06/05/pwn2own-2018-exploit-development/
- **In the last one, they offered a Binary Ninja Commercial License for an exploit**
  - Using only CVE-2018-4193
  - Achieving WindowServer code exec in < 10s
  - Without crashing it
  - With a 90+% reliability
- **Challenge accepted :)**

The bug

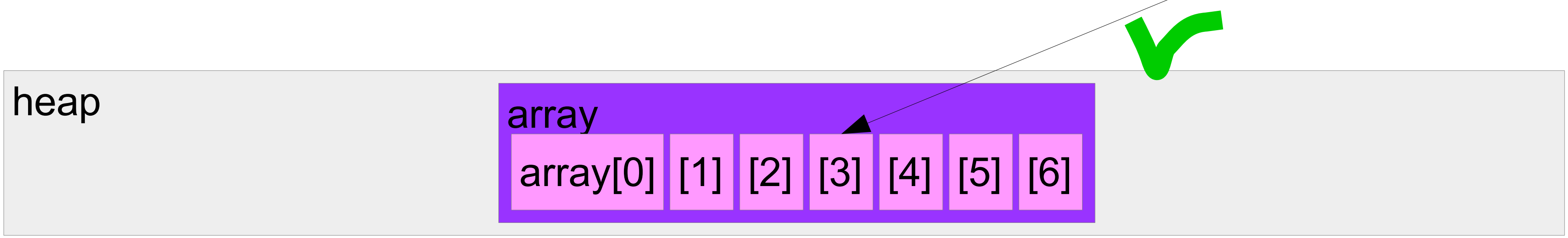
- **Very simple bug**
  - Found via in-process dumb fuzzing
- **Attacker controlled signed index used without lower bound**



The bug

- **Very simple bug**
  - Found via in-process dumb fuzzing
- **Attacker controlled signed index used without lower bound**

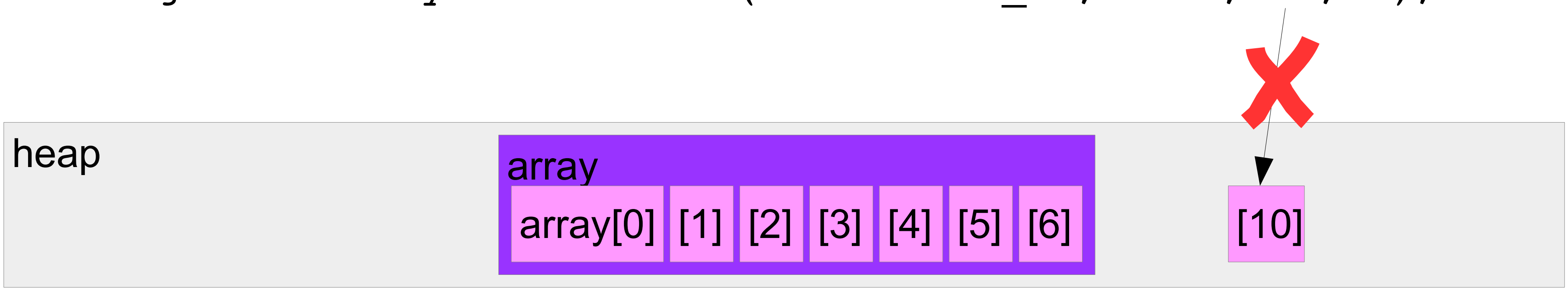
```
SLPSRegisterForKeyOnConnection(connection_id, NULL, 3, 1);
```



The bug

- **Very simple bug**
  - Found via in-process dumb fuzzing
- **Attacker controlled signed index used without lower bound**

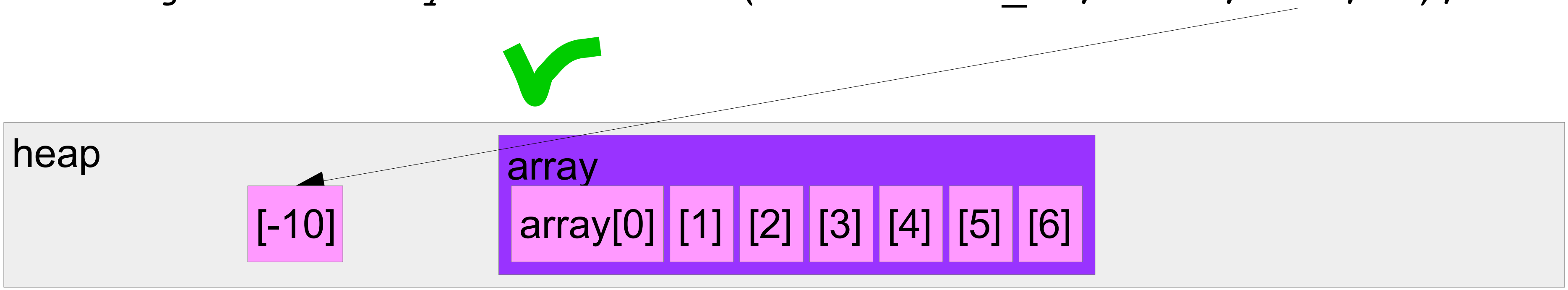
```
SLPSRegisterForKeyOnConnection(connection_id, NULL, 10, 1);
```



The bug

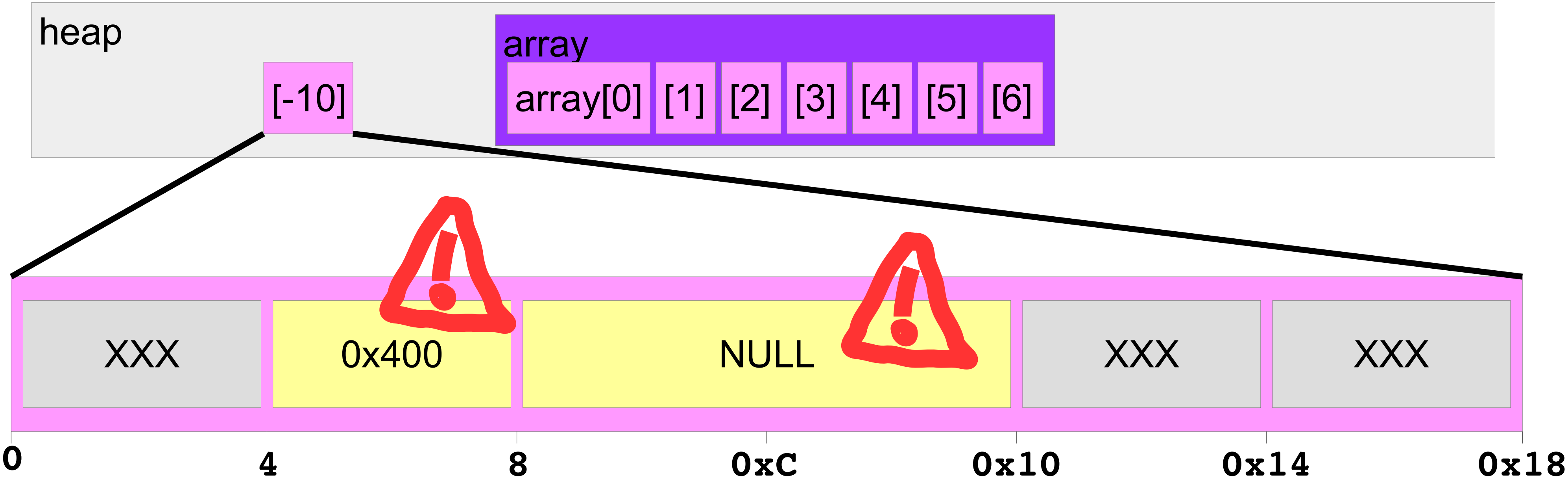
- **Very simple bug**
  - Found via in-process dumb fuzzing
- **Attacker controlled signed index used without lower bound**

```
SLPSRegisterForKeyOnConnection(connection_id, NULL, -10, 1);
```



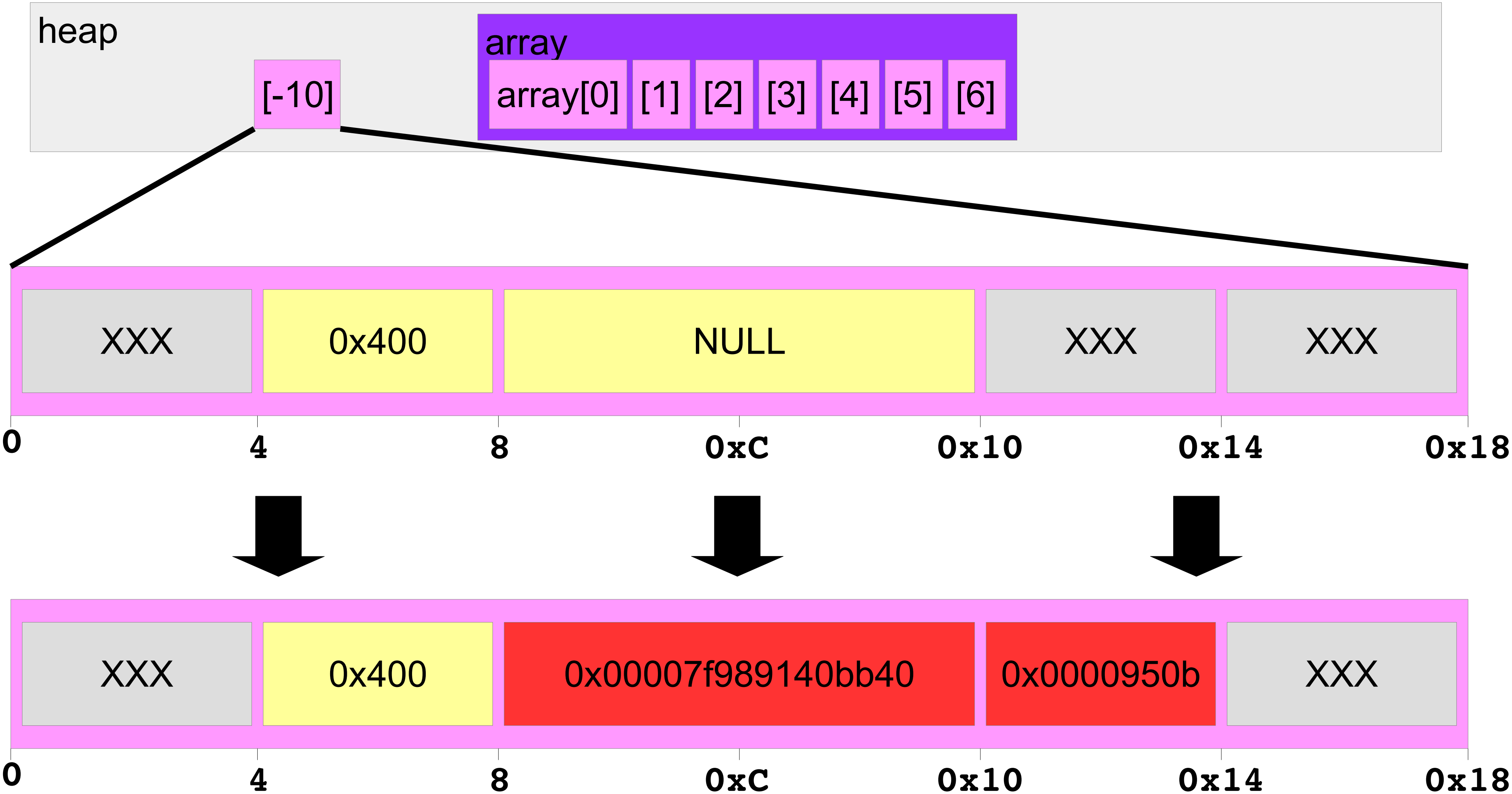
So what can we do?

```
SLPSRegisterForKeyOnConnection(connection_id, NULL, -10, 1);
```

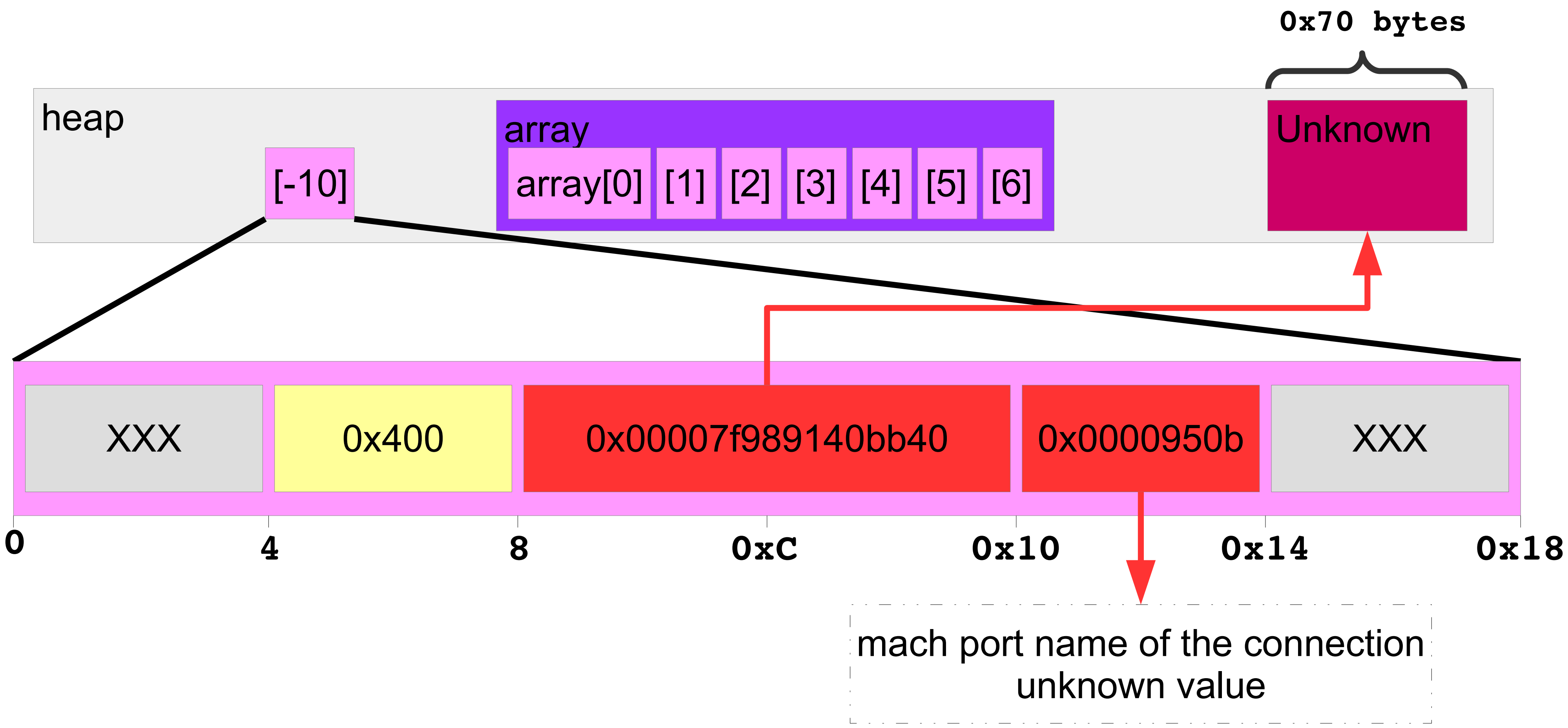


So what can we do?

```
SLPSRegisterForKeyOnConnection(connection_id, NULL, -10, 1);
```



So what can we do?



A little digression… 1/3

- **What's a mach port name?**
  - ID used to identify a port right in a name space
  - Actually an index (24bits) and a gencount (8bits)

    `mach_port_name = (index << 8) | gencount`
  - The gencount is changed (+4) each time an index is reused

    3, 7, 11, …, 247, 251, 3, 7, ...

    Ease the detection of **unintentional** port reuse
  - The name space mach port name table grows when needed

    16, 32, 64, …, PAGE_MAX_SIZE*8, PAGE_MAX_SIZE*16, PAGE_MAX_SIZE*24, etc.
- **Used to be 100% deterministic**
  - Easy to predict `mach_task_self()` value
  - Easy to spray mach port names in the victim name space and to hardcode an attacker controlled mach port name
  - Easy to reuse port names

    Just reallocate them 64 times to make the gencount wrap

    See Brandon Azad blanket exploit

A little digression… 2/3

- **Since iOS 11, Apple decided to *fix* this**
- **Name space freelist is "randomized"**
  - First entry of the new table is always at the beginning of the free list
  - Next, entries are randomly added from the beginning or the end of the table
    - Exactly like the kernel heap
  - The first 8 entries are not yet randomized for compatibility reason
    - `mach_task_self()` is still equal to `0x103` :)
- **Gencount is """"randomized""""**
  - Still initialized with 3
  - Still incremented by 4
  - **BUT** randomly cycle after 16, 32, 48 or 64 generations
    - Instead of 64 before…

A little digression… 3/3

- **So…**
- **`mach_task_self` is still always equal to 0x103**
  - For the moment…
- **It is still possible to spray mach port names and guess there values**
  - Just have to use all the freelist
  - gencount always starts with 3
- **Only problem is for mach port name reuse**
  - We don't know how many time we need to reuse the port to get the same gencount
  - But, if we have an oracle, we can just repeatedly try to reallocate it until it is reused

Recap

- **+++**
  - We can overwrite a NULL pointer with a pointer in the heap
  - We can overwrite a DWORD with a mach port name
- **---**
  - Pointer must be previously NULL
  - Pointer must be prefixed by a 0x400 DWORD
  - We don't know the mach port name (nor the pointer) values

Ret2 Systems exploit

- **Leak the mach port name and pointer values by overwriting a string object**
- **Use the mach port name to overwrite a pointer and gain code execution**
  - Actually a lot more complicated, read their blog post :)
- **Complicated because of the mach port**
  - Last 2 bits are always set → Obj-C tagged pointer
    - Not interesting from an exploitation point of view
    - Cannot directly overwrite Obj-C pointers
  - Mach ports values are low
    - Remember, they are incremental indexes
- **Depending on the heap start address… you might need to spray a lot**
  - Worst case scenario: 4GB
  - OK but can be very slow

Exploit strategy

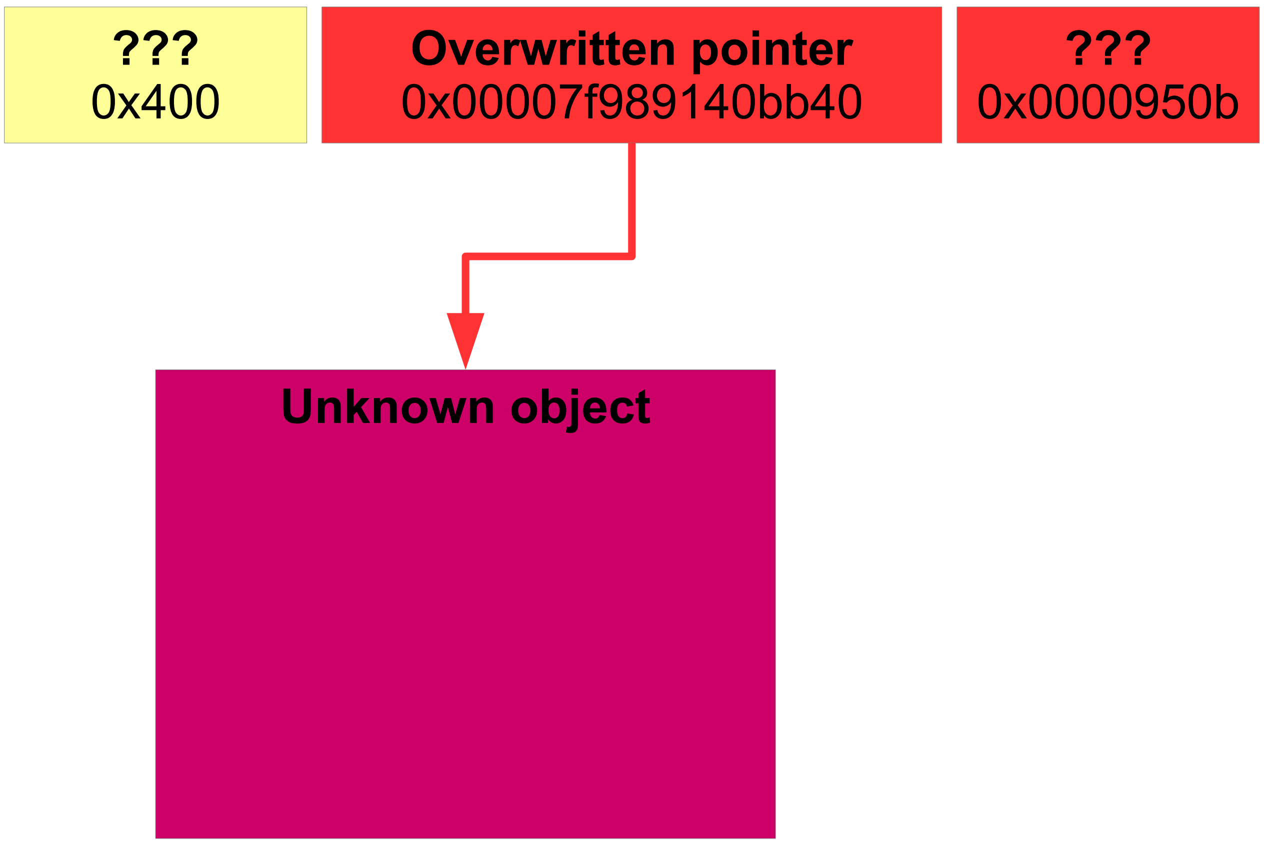
Exploit strategy

- **Why not using the pointer value?**
  - Instead of the mach port name
- **Idea:**
  - Step 1: Overwrite a NULL optional pointer with our unknown heap pointer
  - Step 2: Free the associated object
  - Step 3: Reuse the allocation with controlled data
  - Step 4: Trigger the use of the overwritten pointer to gain arbitrary code execution
  - Step 5: Execute our payload and ensure continuation of execution
- **Easy!**

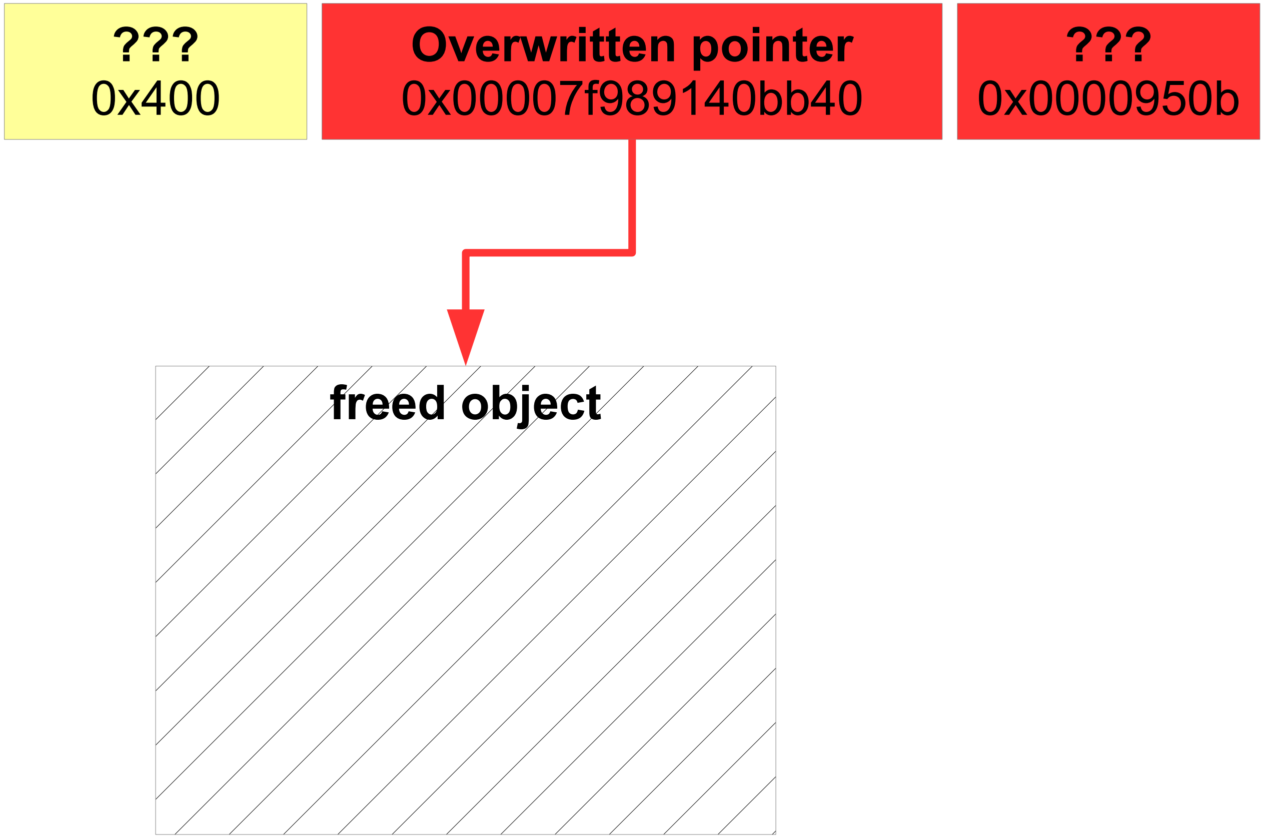
Exploit strategy

???	Optional pointer	???
0x400	NULL	XXX

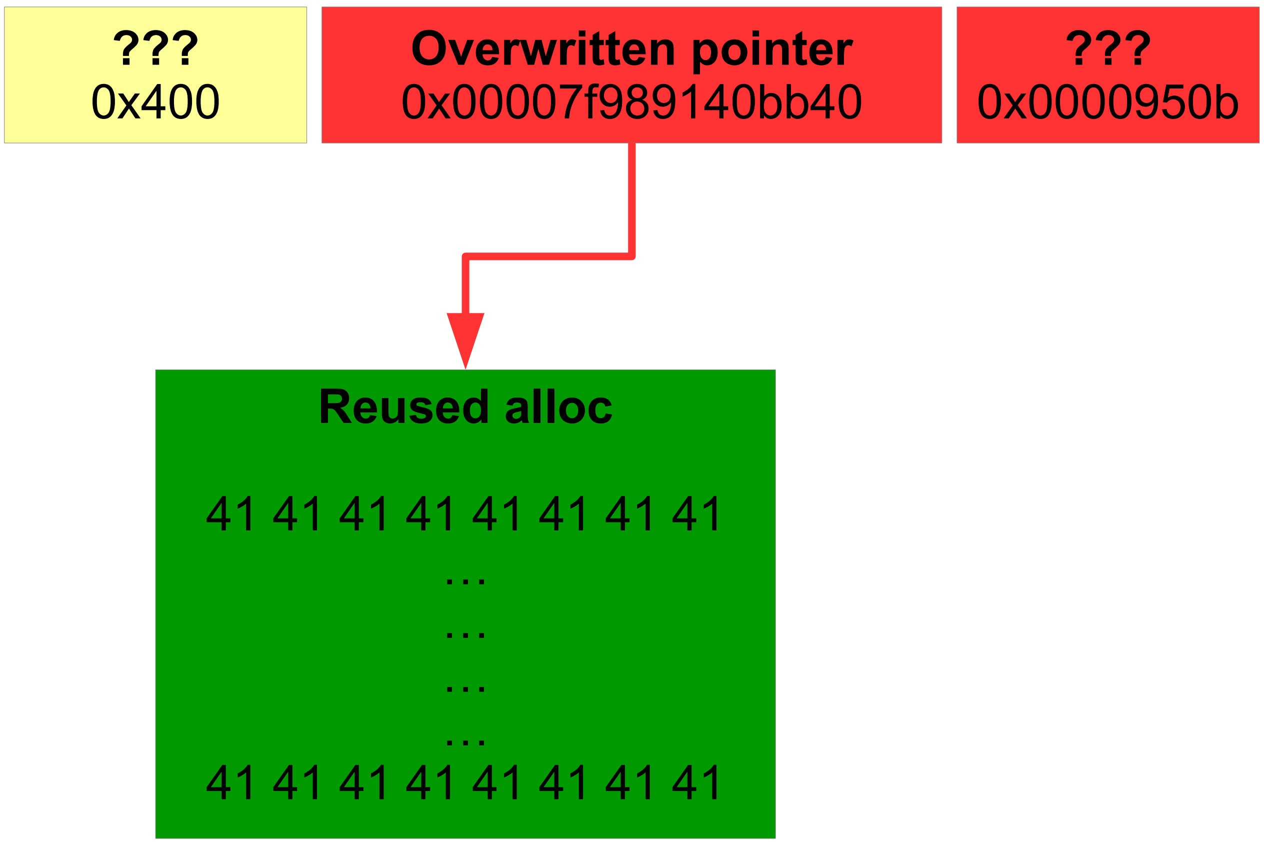
Exploit strategy



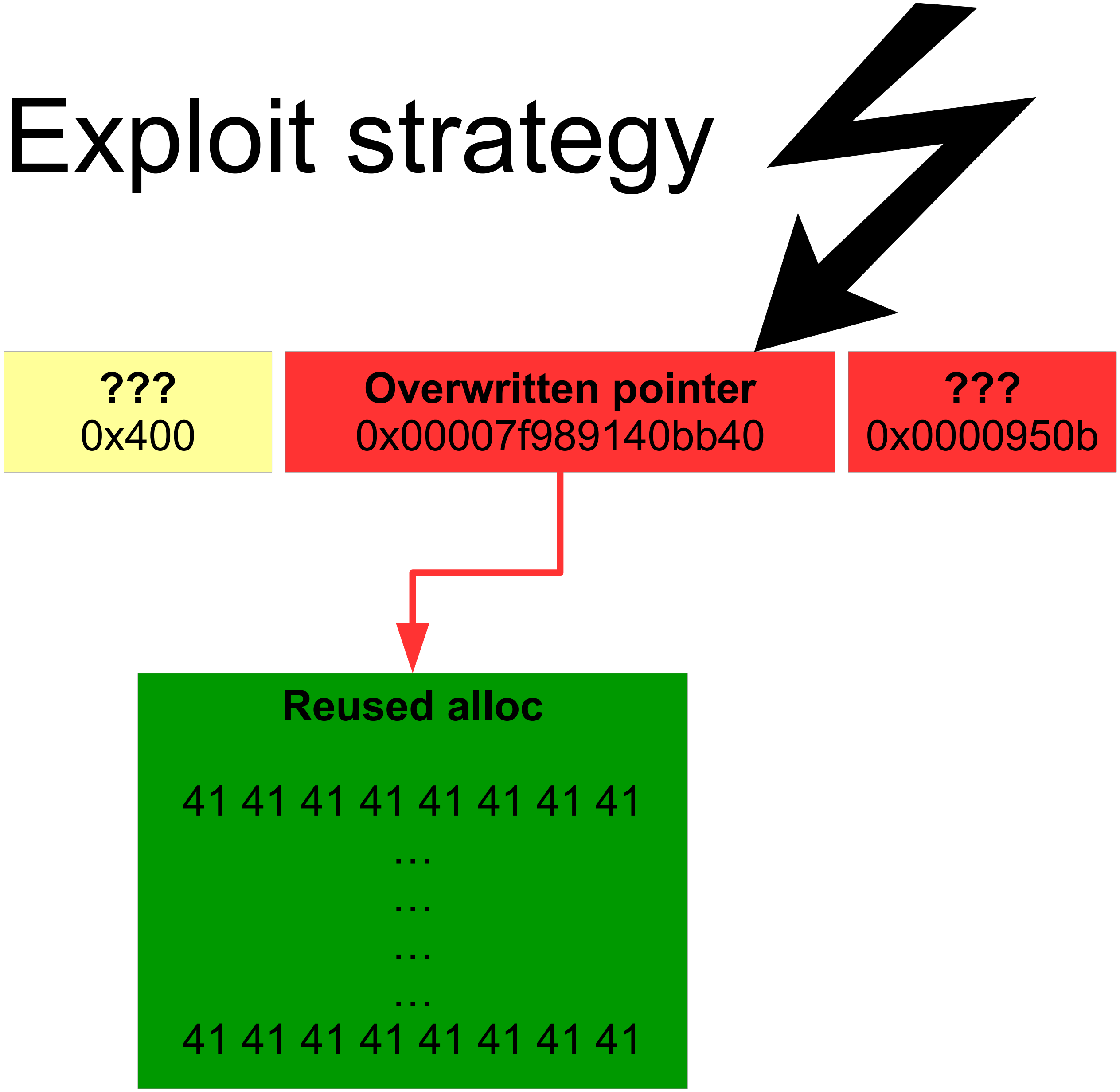
Exploit strategy



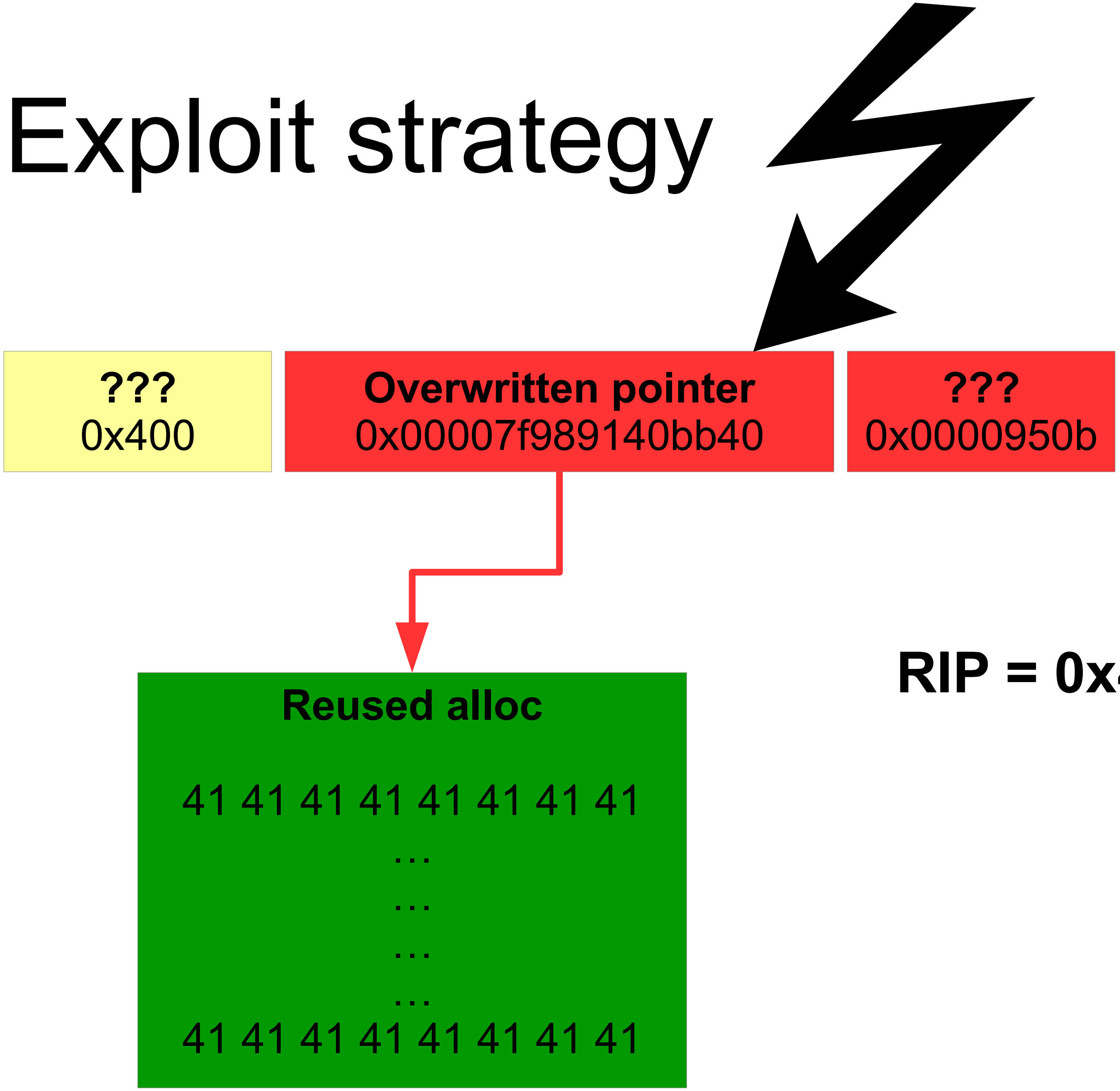
Exploit strategy



Exploit strategy

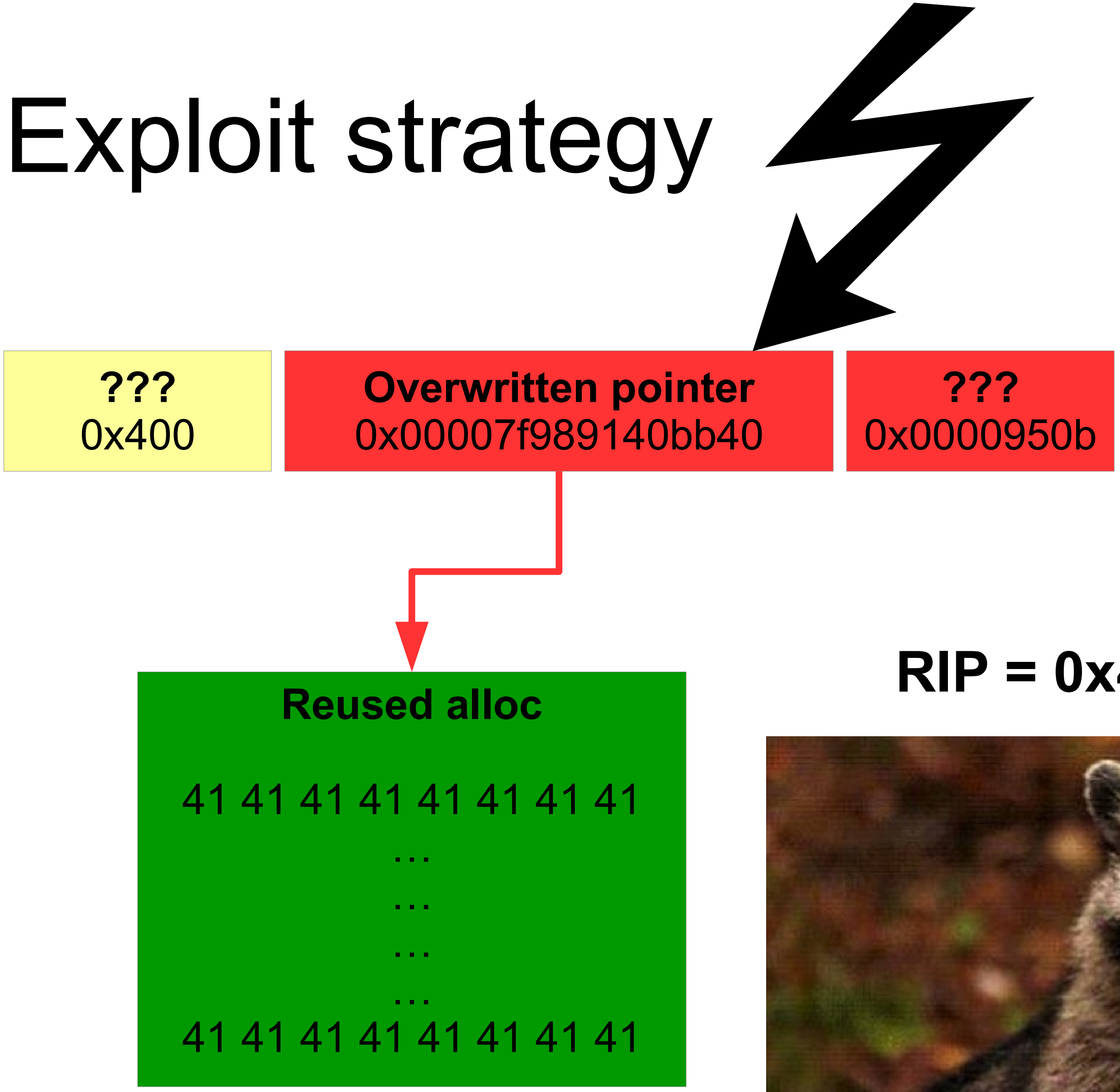


Exploit strategy



RIP = 0x4141414141414141

Exploit strategy



RIP = 0x4141414141414141

Step 1

Overwrite a NULL optional pointer with our unknown heap pointer

Tools

- **WindowServer gives us a powerful primitive**
- **`SLSSetConnectionProperty`:**
  - Can be used on any connection, no privilege required
  - Arbitrary property name
  - Property value are Obj-C values deserialized from the user input
  - We can read, modify and delete properties
- **Objects are deserialized via `CFPropertyListCreateWithData`**
  - From the doc:

    `CFPropertyListRef` can be any of the property list objects: `CFData`, `CFString`, `CFArray`, `CFDictionary`, `CFDate`, `CFBoolean`, and `CFNumber`.
- **Convenient way to:**
  - Massage the heap
  - Read back modified properties
  - Place arbitrary data in memory

Problem

- **We need to have the following shape:**
  - `0x400 | NULL | DWORD`
  - Where `DWORD` can be safely overwritten with a mach port name
  - Where the `NULL` pointer, once overwritten, will be used
- **We can allocate arbitrary**
  - `CFData`
  - `CFString`
  - `CFArray`
  - `CFDate`
  - `CFBoolean`
  - `CFNumber`
  - `CFDictionary`

Let see what we can do… 1/2

- **`CFData` and `CFString` can be used to leak**
  - Put `\x00\x04\x00\x00\x00\x00\x00\x00\x00\x00` in the string/data to get the pre-condition
  - Read it back after triggering the vulnerability
  - Used by ret2 to get mach port name and pointer values
  - Cannot be used to get code exec…
- **`CFArray` cannot be used**
  - "`NULL`" Obj-C pointers aren't actually `NULL` but are the singleton `kCFNull`

    And `kCFNull` is not serializable anyway…
  - Pre-condition cannot be met
- **`CFDate`, `CFBoolean` and `CFNumber` are useless**
  - `CFNumber` are limited in size, 128bits max
  - `CFDate` are just doubles
  - `CFBoolean` are singleton

Let see what we can do… 2/2

- **`CFDictionary`**
  - Use a hash table…
  - Hash tables contain `NULL` pointers
  - During hash table destruction all non `NULL` pointers will be released!
- **Win?**
  - We need to overwrite **two** pointers

    Value without key → crash (NULL deref)

    Key without value → pointer is unused
  - We still need to put a `0x400` before the `NULL` pointer
  - Is it safe to rewrite the `DWORD` after the pointer with a mach port name?
- **We need to go deeper…**
  - Let’s reverse CoreFoundation code!

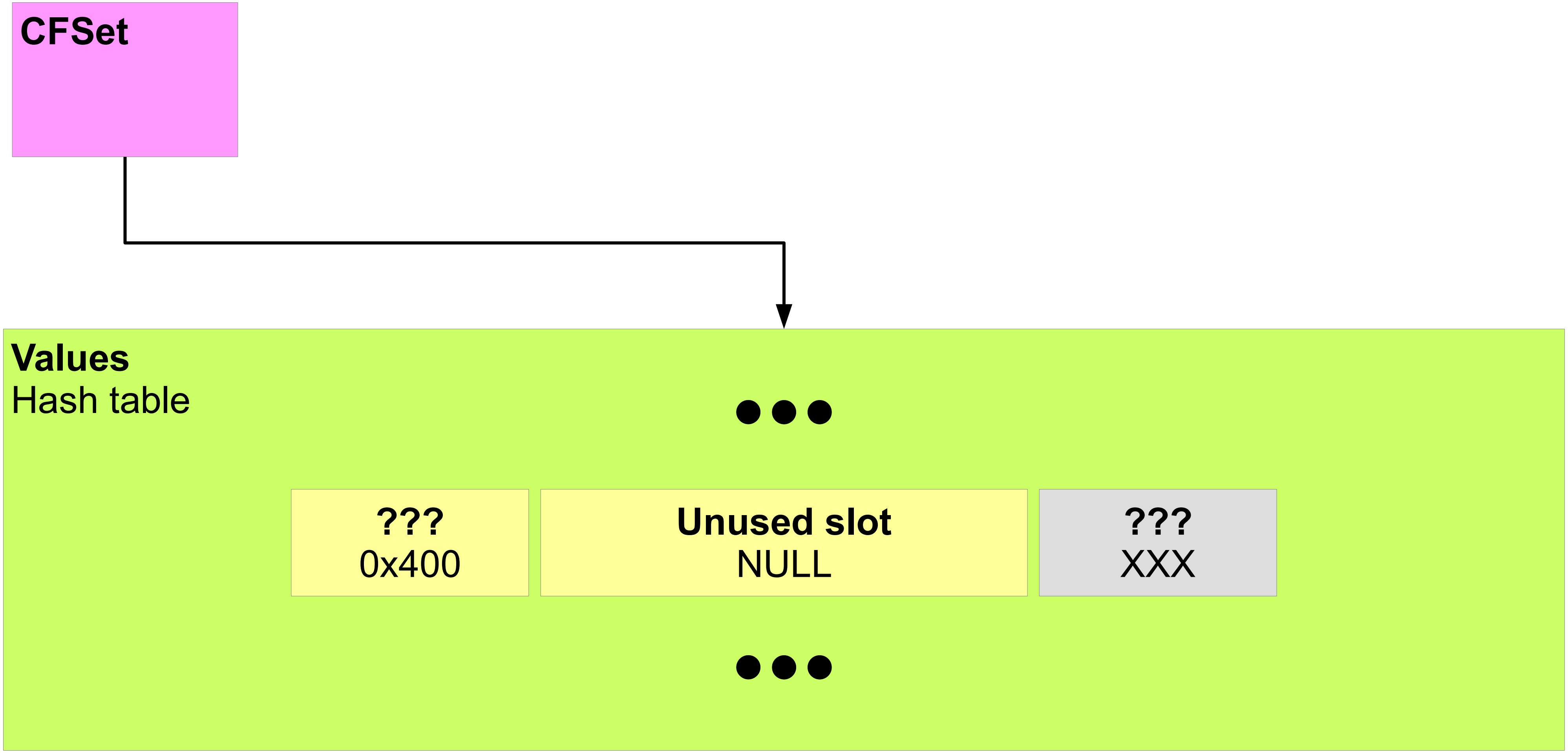
CFPropertyListCreateWithData

- **Accepts 3 different formats**
- **Tries successively to decode**
  - Binary format (bplist0 header)
  - XML
  - Old plist format (Json-like)
- **Let's study/reverse all the implementations!**
  - CoreFoundation is (kinda, no updates for 4 years) open source
  - XML and old format are not really interesting
  - Binary format however…

CFBinaryPlistCreateObjectFiltered

- **Binary format supports more objects than the others**
  - `CFKeyedArchiverUID`
  - `CFNull`
  - `CFSet`
- **`CFNull` and `CFSet` are only supported when deserializing!**
  - We need to forge our own serialized objects
  - Fortunately for us it's not that complicated…
- **`CFSet` gives code exec with a single overwritten pointer!**
  - Same hash table structure than `CFDictionnary`
  - But obviously with only values and no keys
- **We still need to put a `0x400` before our `NULL` pointer…**

Battle plan



CoreFoundation internals

- **Some objects are stored directly in their reference**
  - The reference isn't a pointer anymore but directly the value
    - Obviously only work for small values
  - Saves some memory and CPU cycles
- **To identify those objects, their lowest significant bit is set**
  - Because heap pointers are always 16 bytes aligned
- **The next three lowest significant bits encode the type**
  - `NSAtom`→0, `CFString`→2, `CFNumber`→3, `NSIndexPath`→4, `NSDate`→6
  - Warning: tagged types are lazy initialized
    - WindowServer only use `CFString`, `CFNumber` and `NSAtom`
    - We can also force it to use `NSDate` by deserializing dates but not the others

CFNumber and CFString

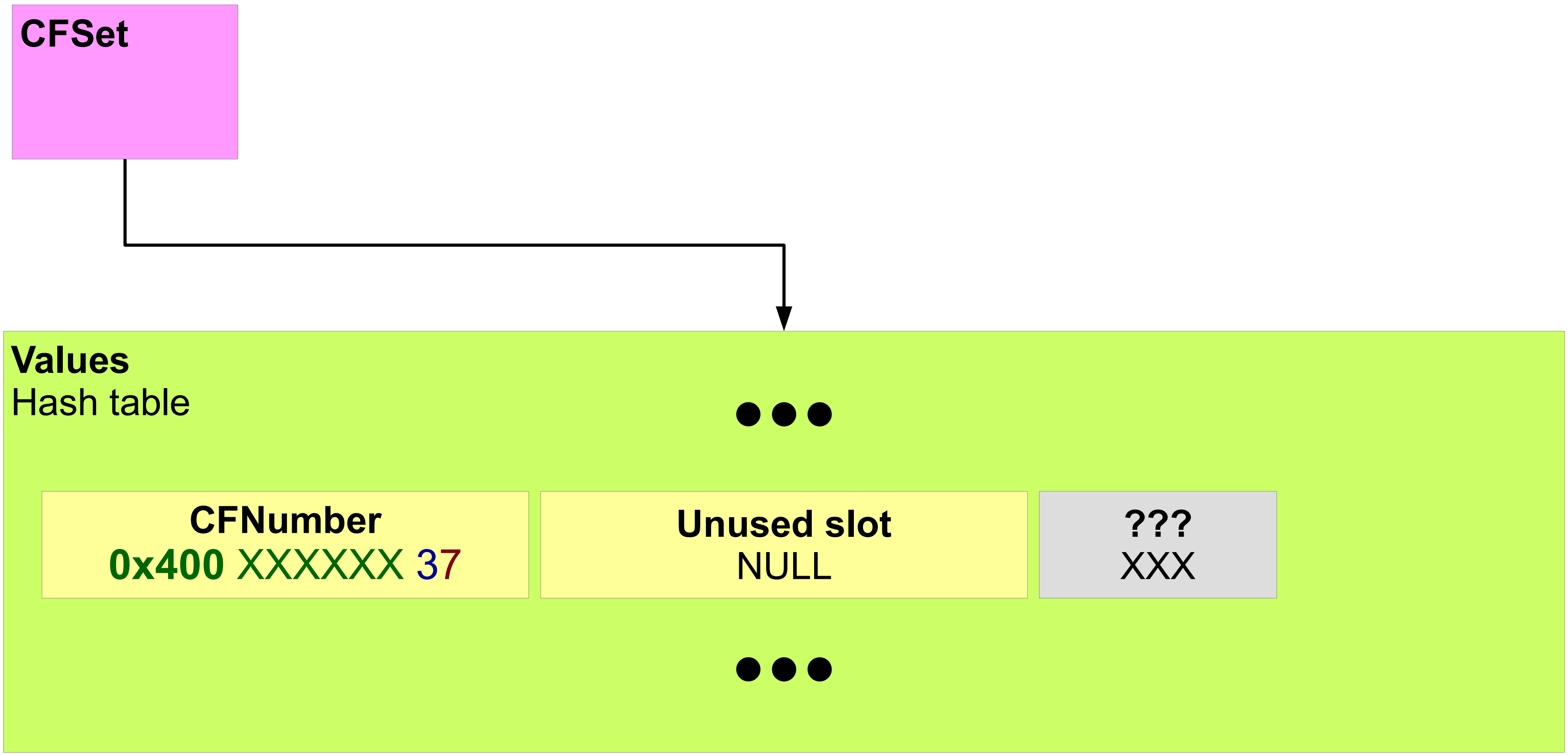
0x1122334455667737

Value	Type	Class
0x11223344556677	QWORD	CFNumber

0x4142434445464775

Value	Length	Class
“GFEDCBA”	7	CFString

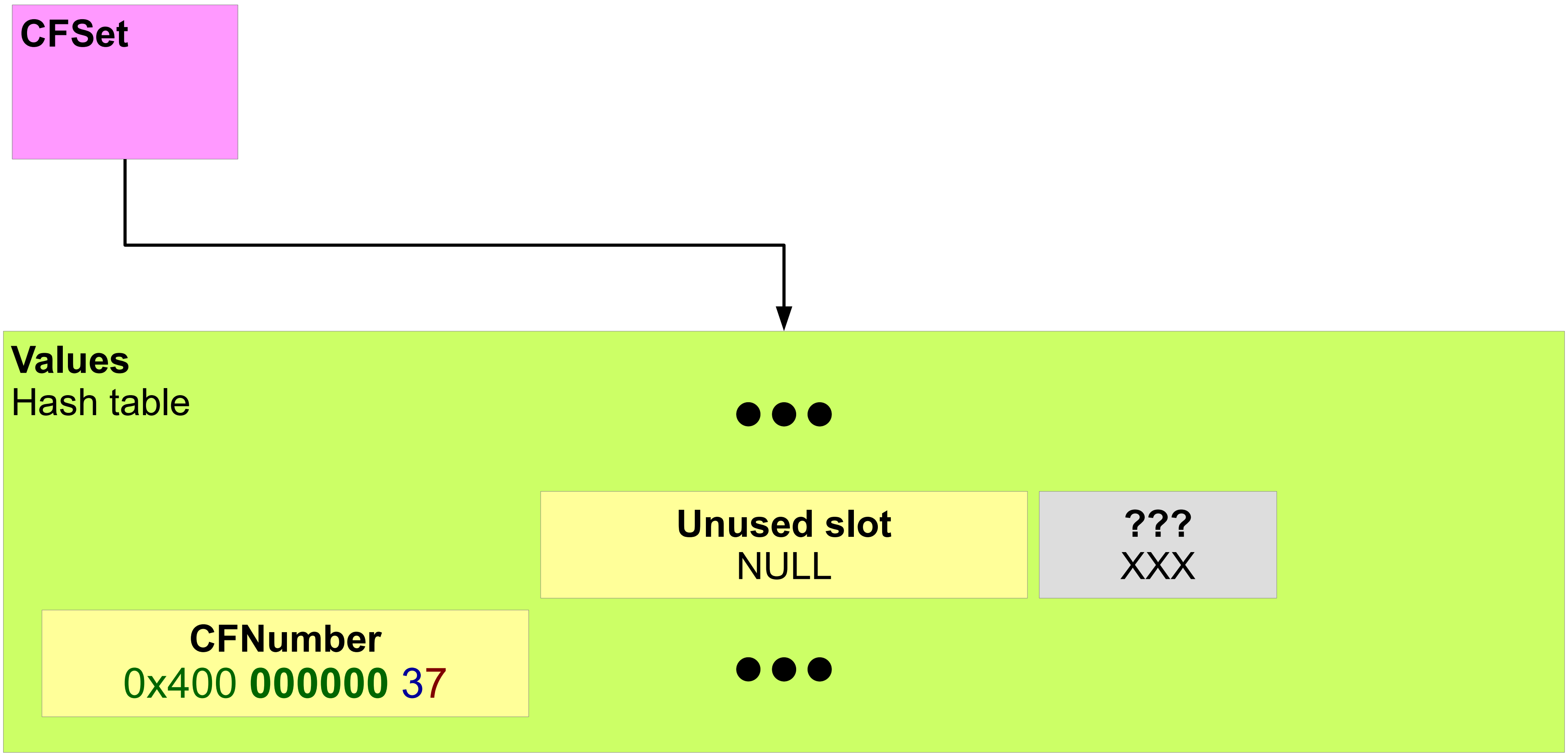
Battle plan


CFSet
Values
Hash table
CFNumber
0x400 XXXXXX 37
Unused slot
NULL
???
XXX

How do we place our values?

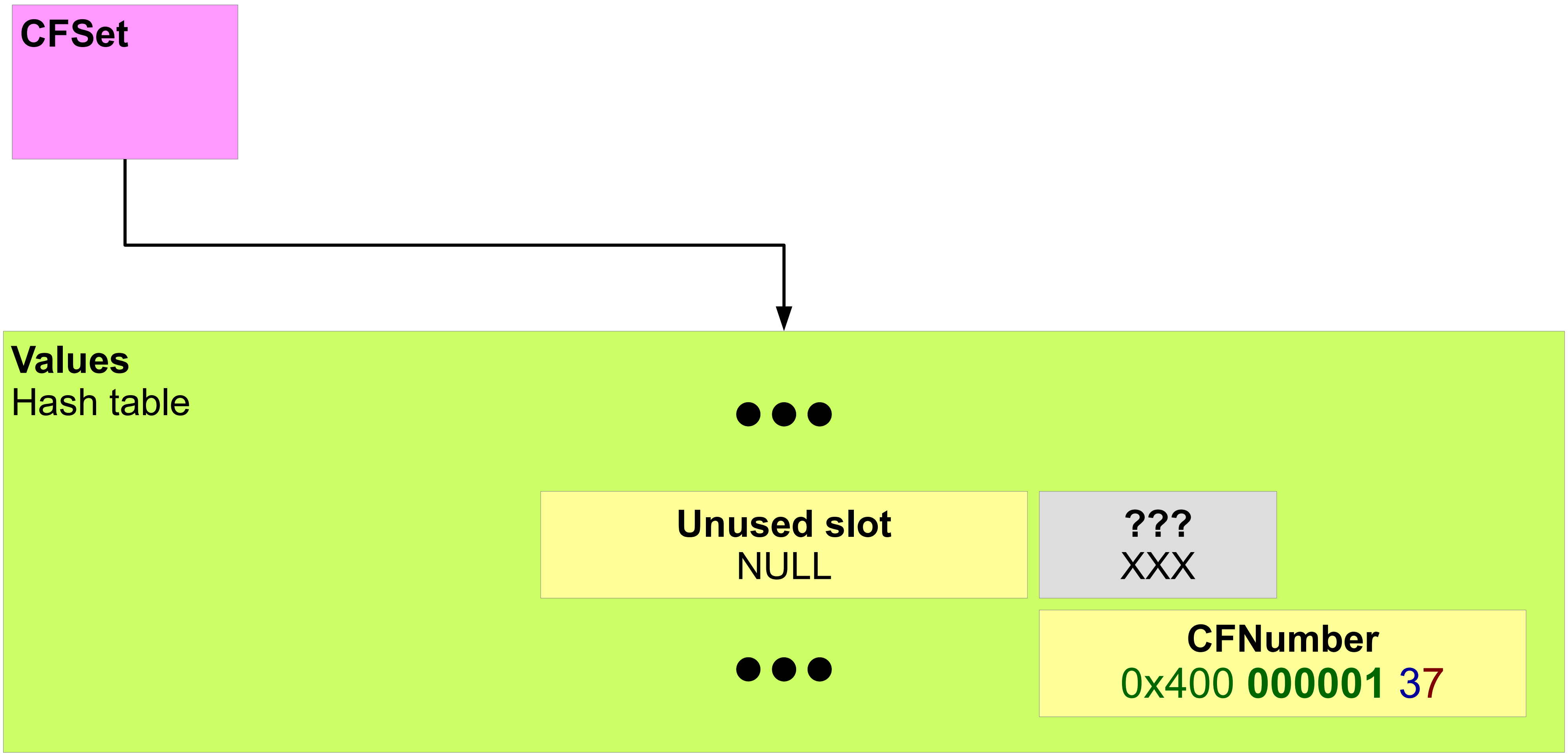
- **`CFSet` is a generic construction and can be used with any type**
- **Callbacks must be passed during `CFSet` creation**
  - hash, equal, release, retain
- **`kCFTypeSetCallBacks` are the built-in callbacks for `CFTypes`**
  - hash → `CFHash`
  - equal → `CFEqual`
  - retain/release → wrappers around `CFRetain`/`CFRelease`
- **CFHash is deterministic!**
  - We can precisely place our `CFIntegers` in the hash table
  - We just change the less significant bits of the `CFInteger` until it is correctly placed in the hash table
- **We can put (almost) arbitrary `QWORDs` in `CFDictionary` and `CFSet` hash tables**
  - Only lowest significant bytes are not 100% controlled
  - Not a problem as we only need to control the 32 highest bits

Battle plan

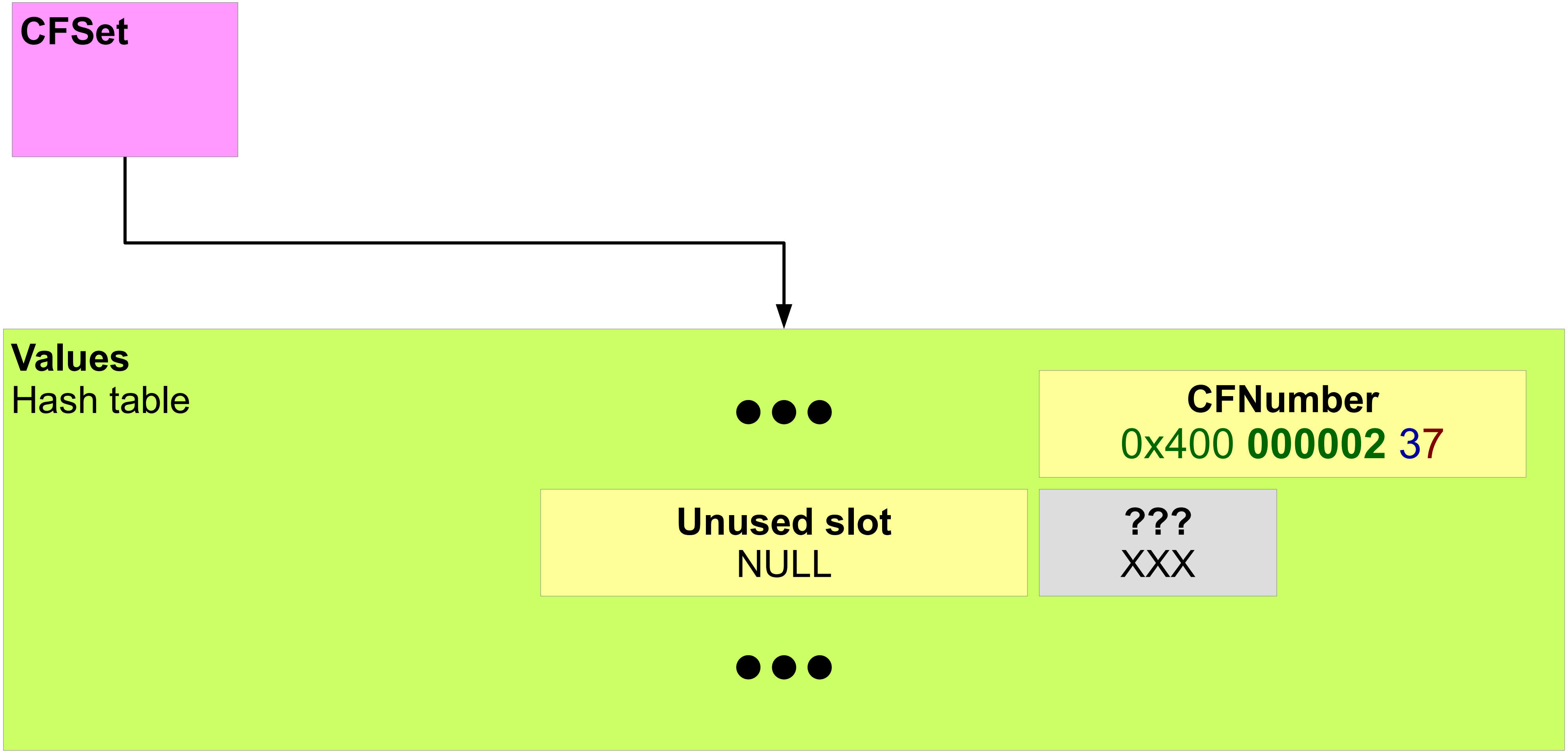


CFSet
Values
Hash table
Unused slot
NULL
???
XXX
CFNumber
0x400 000000 37

Battle plan

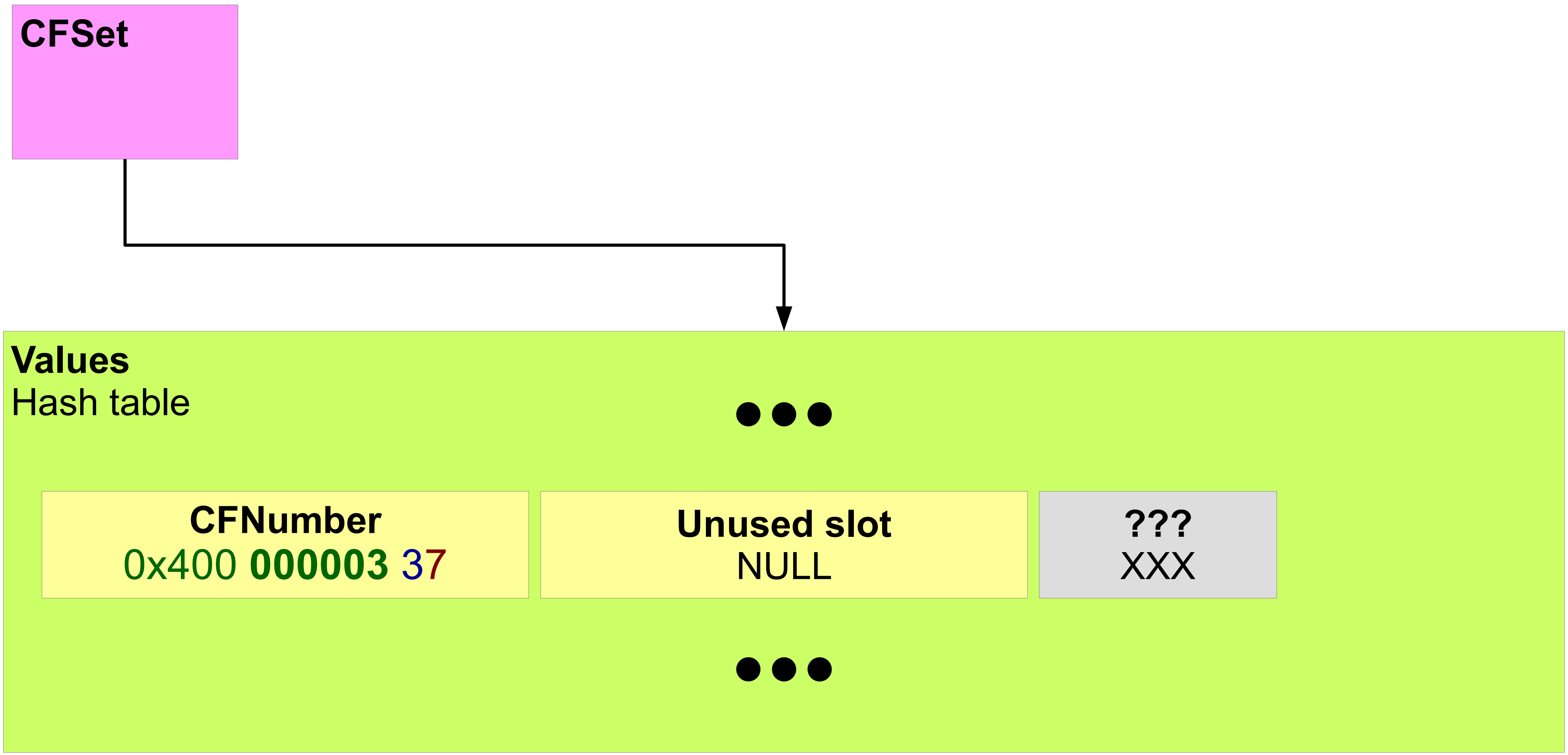

CFSet
Values
Hash table
Unused slot
NULL
???
XXX
CFNumber
0x400 000001 37

Battle plan



CFSet
Values
Hash table
CFNumber
0x400 000002 37
Unused slot
NULL
???
XXX

Battle plan

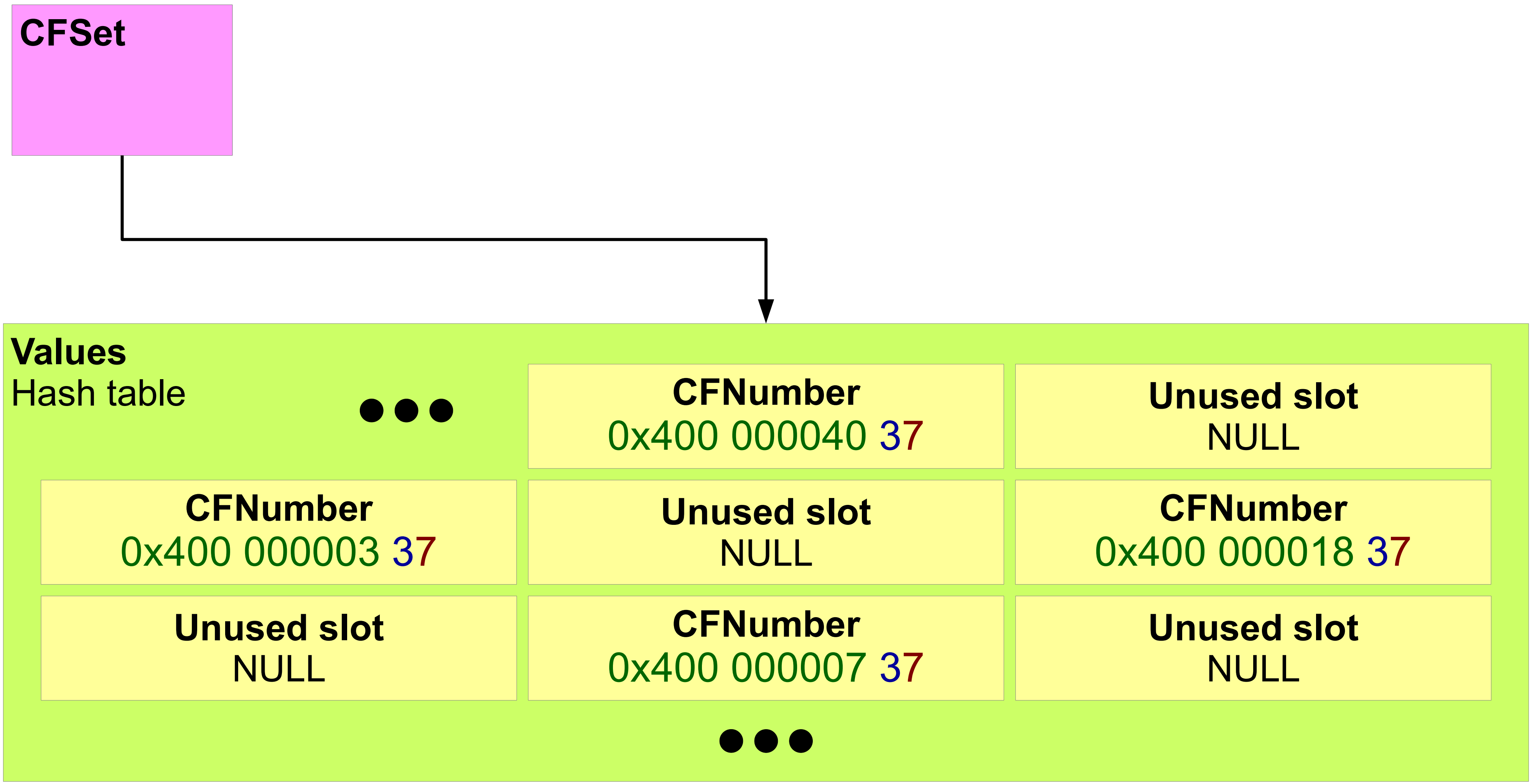


CFSet
Values
Hash table
CFNumber
0x400 000003 37
Unused slot
NULL
???
XXX

What about the mach port name?

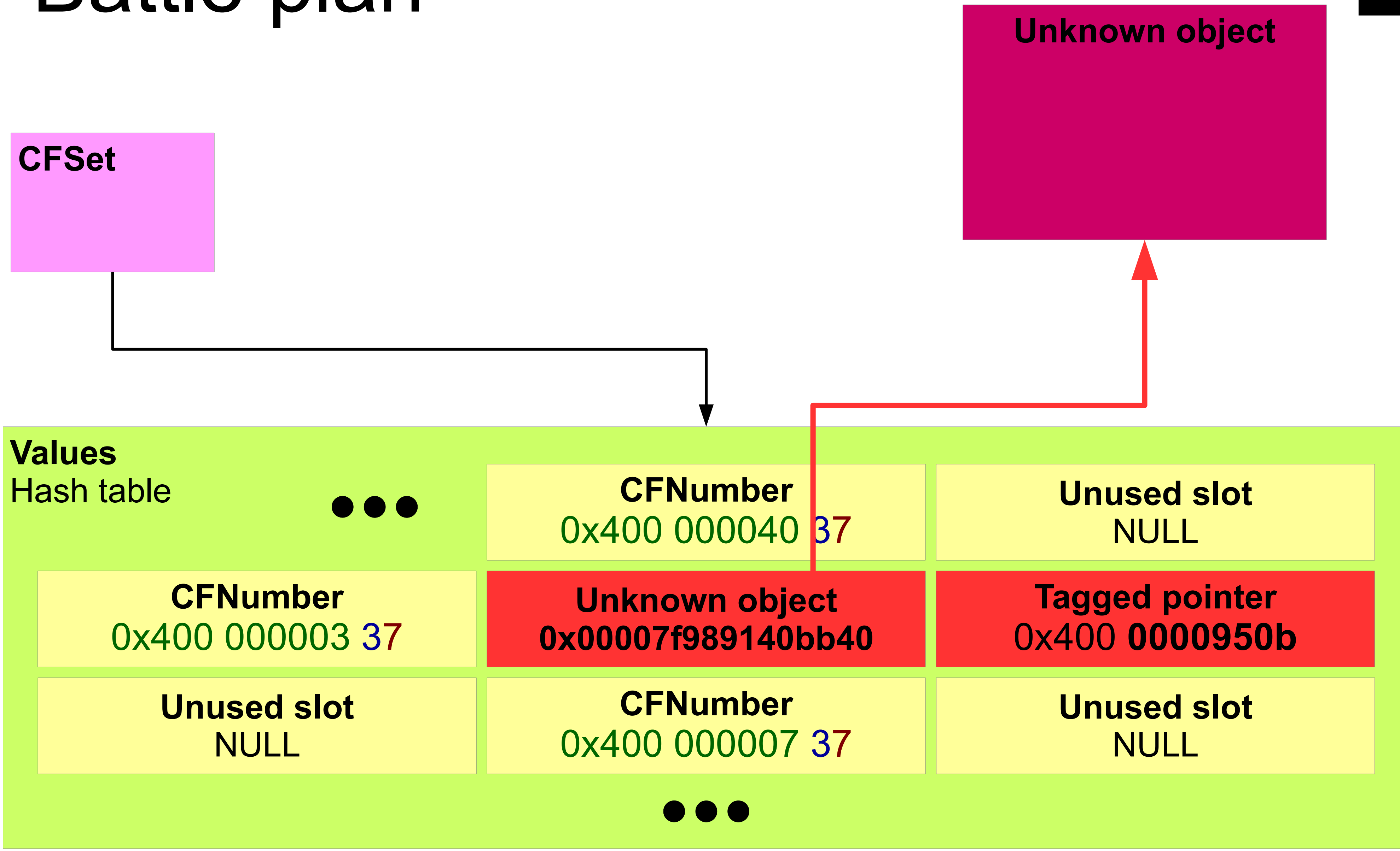
- **But what about the mach port name?**
  - We will have an unknown pointer in our hash table
- **It will be considered as a tag pointer**
  - XXX3 → Invalid tagged type
  - XXX7 → `NSNumber`
  - XXXB → Invalid tagged type
  - XXXF → Invalid tagged type
- **Fortunately for us:**
  - `CFRelease` just do nothing if a pointer is tagged!
  - Regardless of the (valid or invalid) type

Battle plan



CFSet
Values
Hash table
CFNumber
0x400 000040 37
Unused slot
NULL
CFNumber
0x400 000003 37
Unused slot
NULL
CFNumber
0x400 000018 37
Unused slot
NULL
CFNumber
0x400 000007 37
Unused slot
NULL

Battle plan

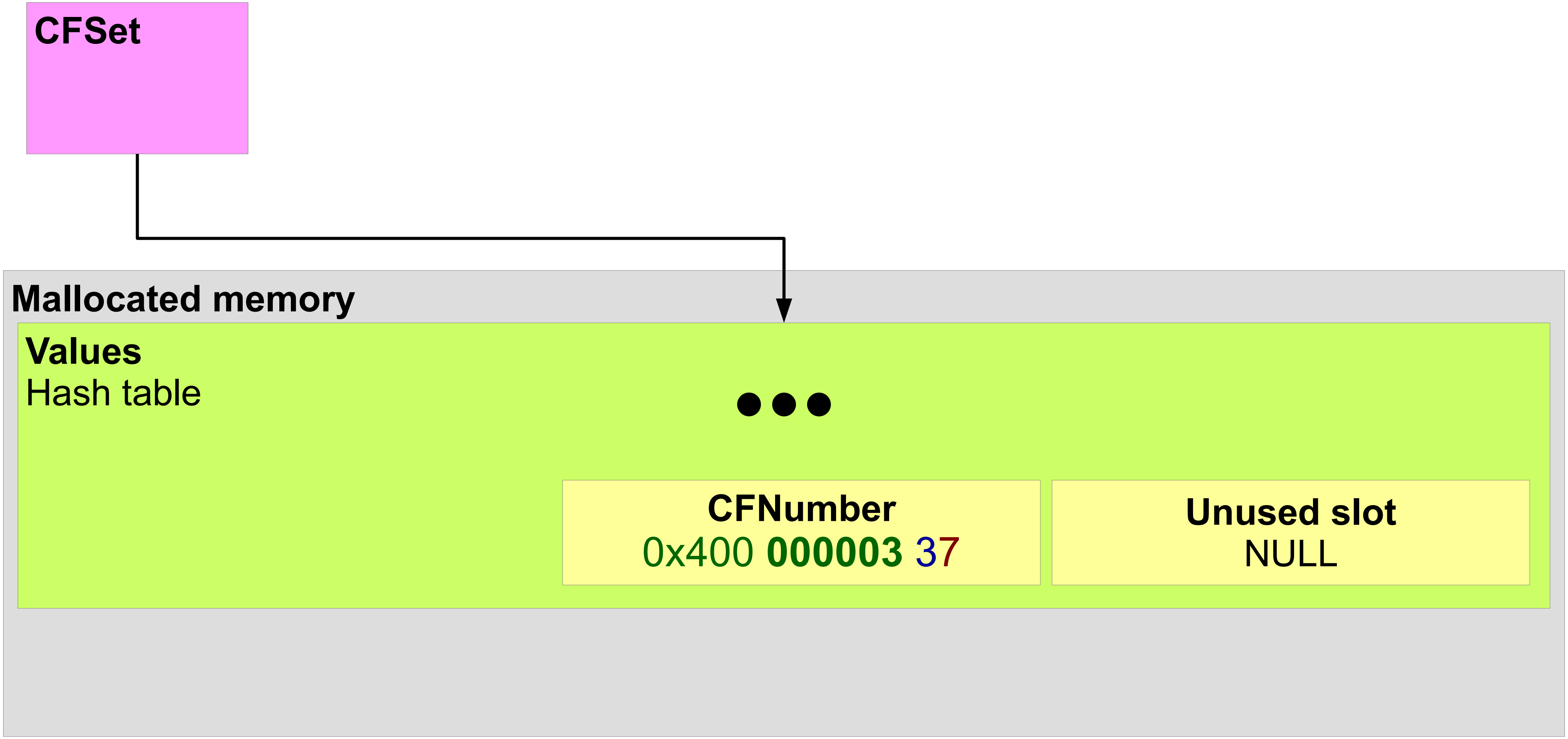


CFSet
Unknown object
Values
Hash table
CFNumber
0x400 000040 37
Unused slot
NULL
CFNumber
0x400 000003 37
Unknown object
0x00007f989140bb40
Tagged pointer
0x400 0000950b
Unused slot
NULL
CFNumber
0x400 000007 37
Unused slot
NULL

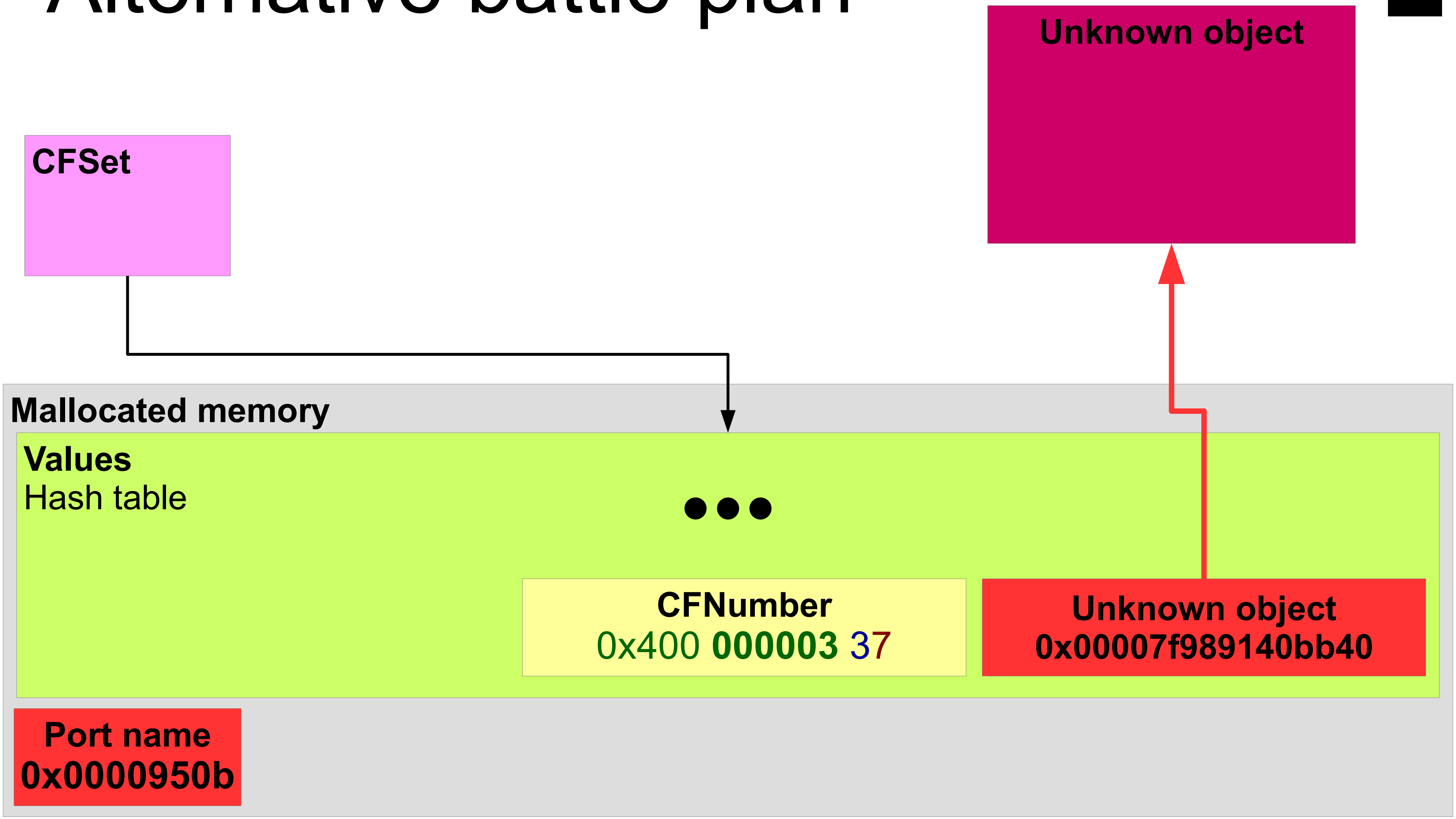
What if the lowest bit wasn't set?

- **Allocations size in the heap are multiples of 0x10**
  - Even 0x200 for "small" allocation (> 0x400)
- **Hash tables elements counts are prime numbers**
  - 3, 7, 13, 23, 41, 71, 127, 191, etc.
- **There is at least 8 unused bytes at the end of every hash table**
  - It is always safe to overwrite them
- **We could just place our 0x400 in the penultimate slot of the hash table**
  - Overwritten pointer will be the last
  - Mach port name will be outside the hash map…
  - …but still in the dedicated allocation → win

Alternative battle plan



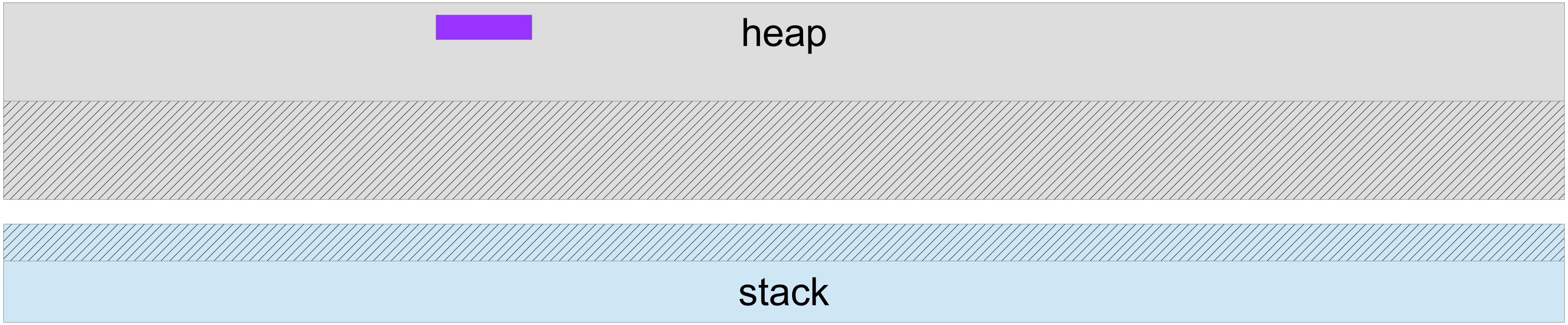
Alternative battle plan



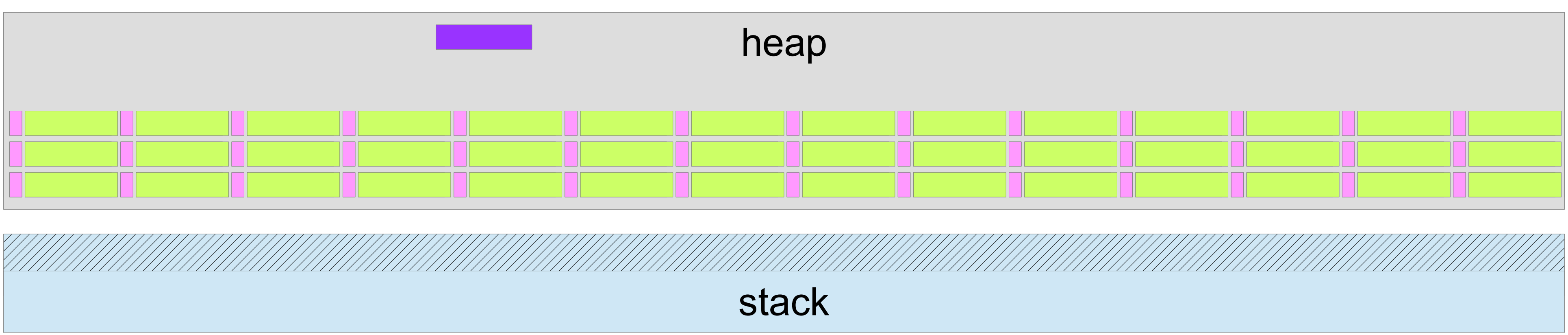
Last problem

- **We can only specify a negative index**
- **And the array is global and early allocated**
  - No way to allocate controlled data before it
- **Fortunately, base of the heap is just before the stack…**
  - See `mvm_aslr_init` in `libmalloc`
- **…and grows backward 256MiB per 256MiB**
  - See `mvm_allocate_pages_securely` in `libmalloc`
- **So if we allocate enough `CFSet`, they will be accessible with a negative offset!**
  - We can do this in one operation by assigning a big `CFArray` of `CFSet` to a given property name

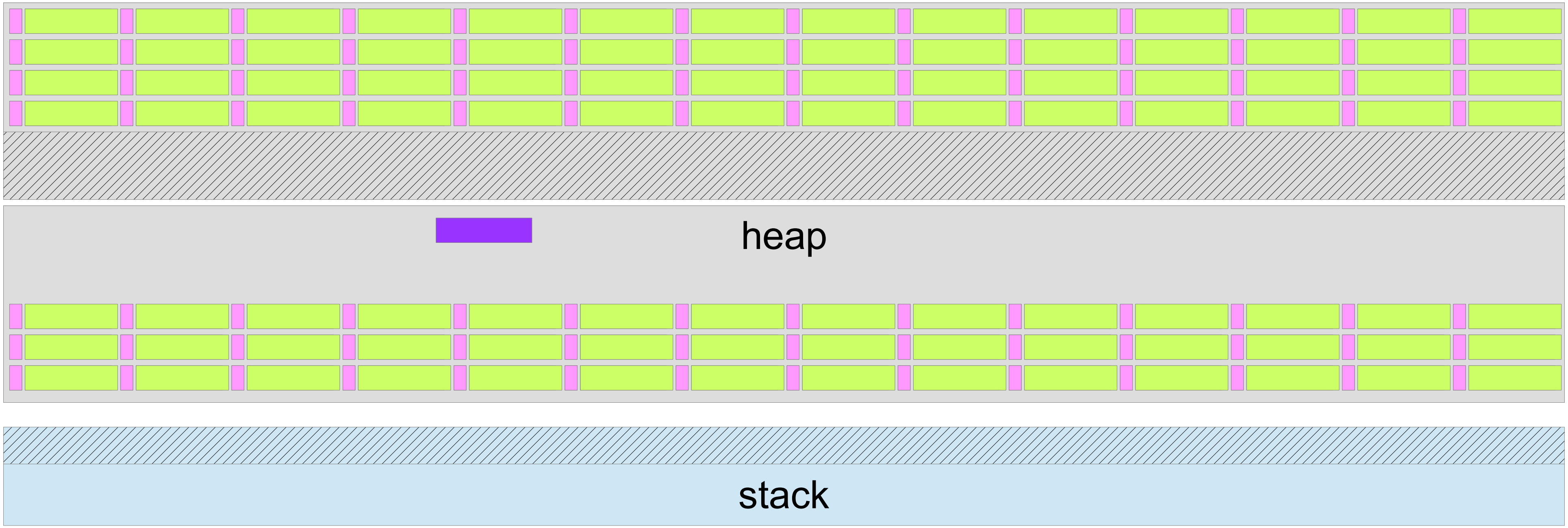
Battle plan



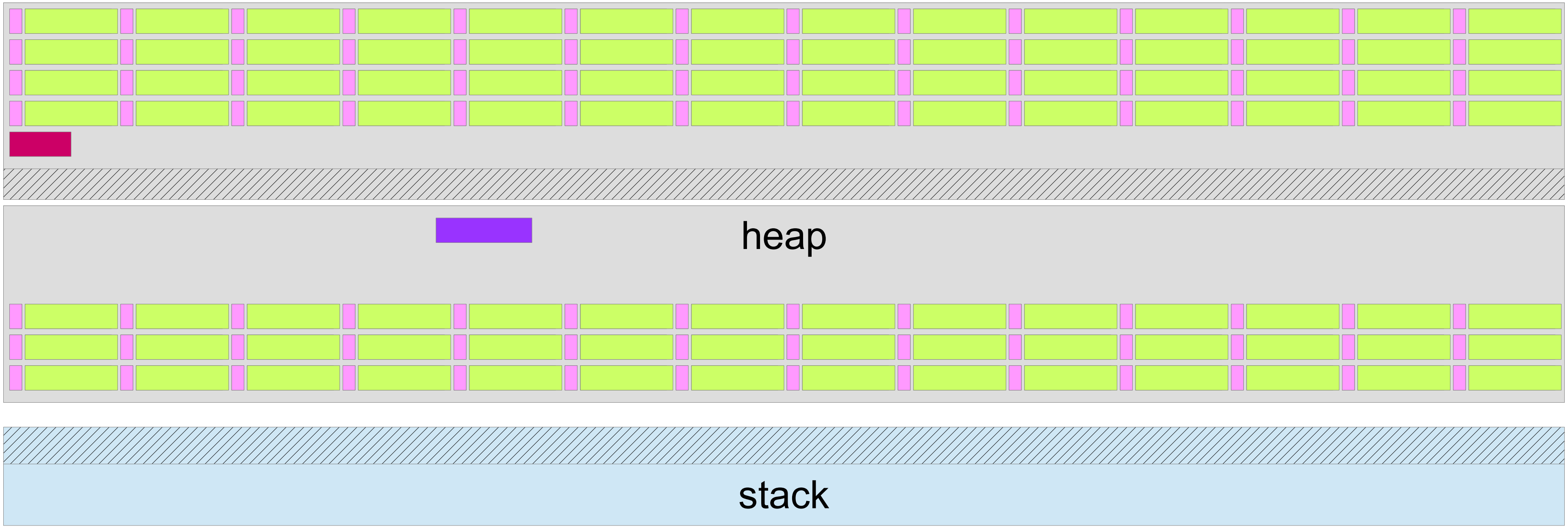
Battle plan



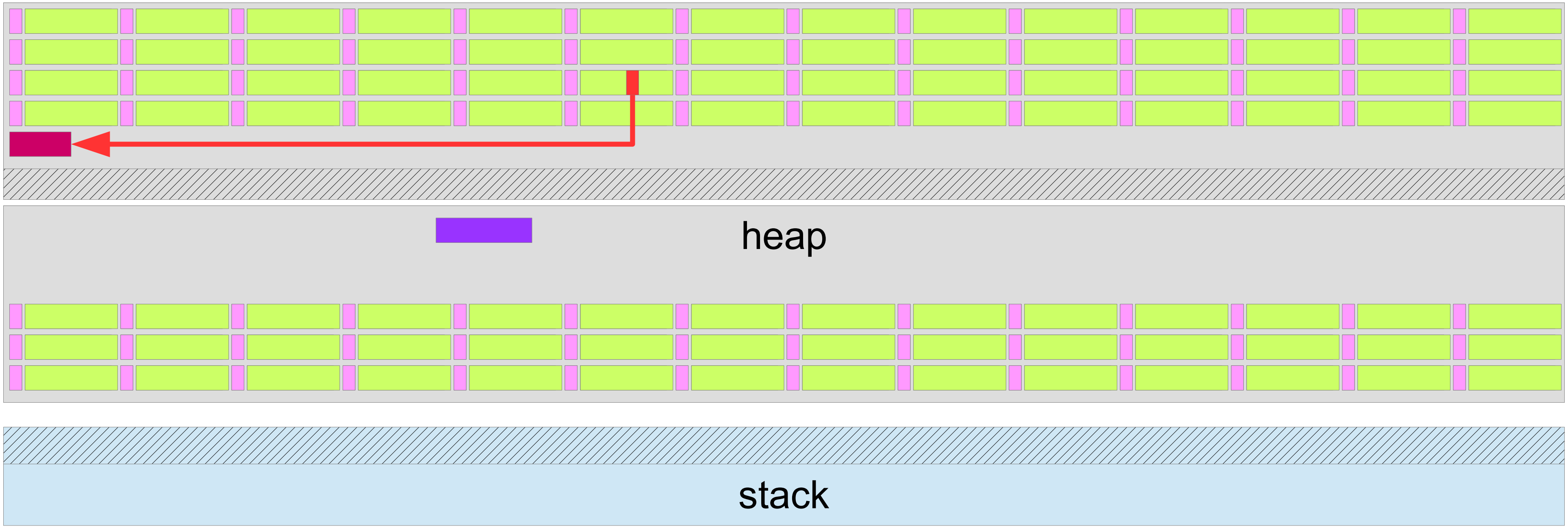
Battle plan



Battle plan



Battle plan



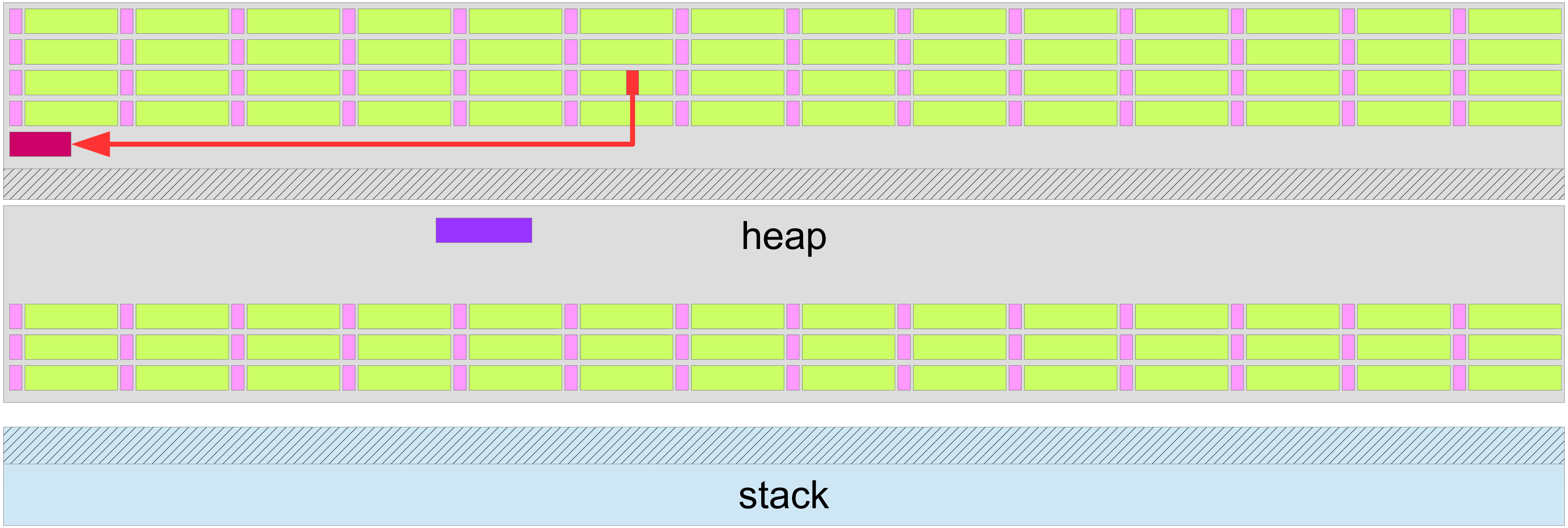
Step 2

Free the associated object

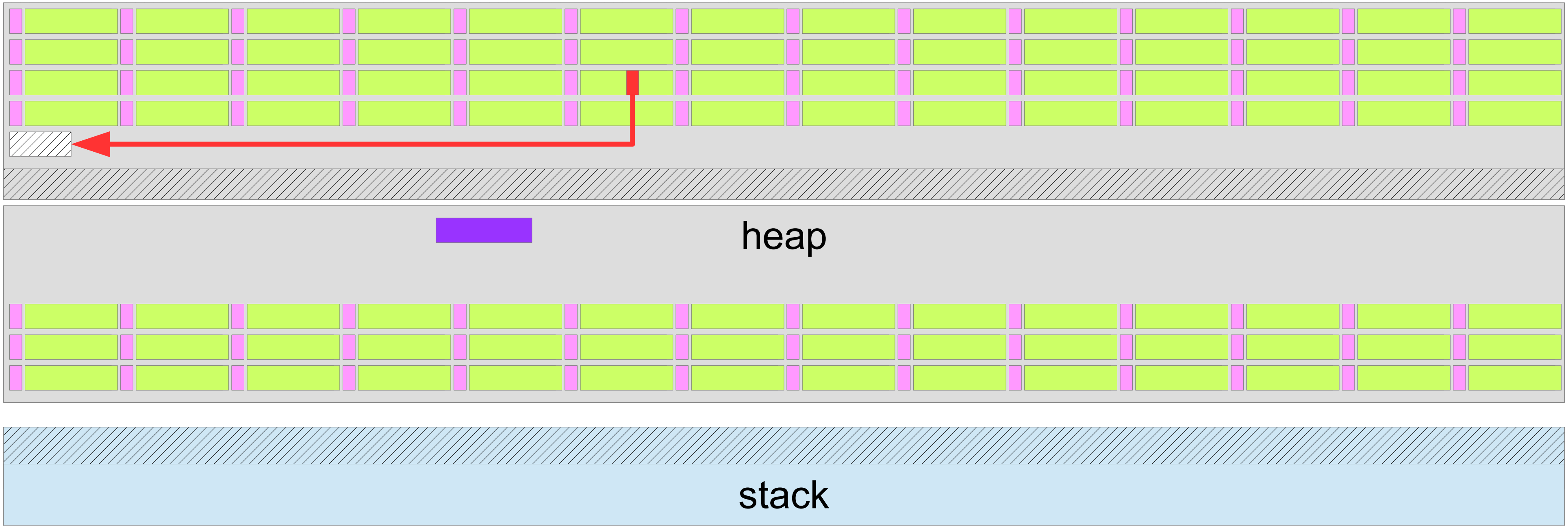
Freeing the object

- **Nothing really interesting there**
  - Just reverse WindowServer and find how to free the object
- **Turns out that the 0x70 bytes object represents an application**
  - It is possible to allocate multiple applications per connection
- **Wasn't that easy…**
  - Not that easy to free an application without killing the whole connection
  - A lot of boring reverse is not included in this presentation :)
- **WindowServer is very complex**
  - There might be other exploitable vulnerabilities…
  - …but I can only use CVE-2018-4193

Battle plan



Battle plan



Step 3

Reuse the allocation with controlled data

CoreFoundation internals

- **Our overwritten pointer will be considered as a pointer on a CoreFoundation object**
  - Same than an Objective-C object
  - See nemo Phrack article

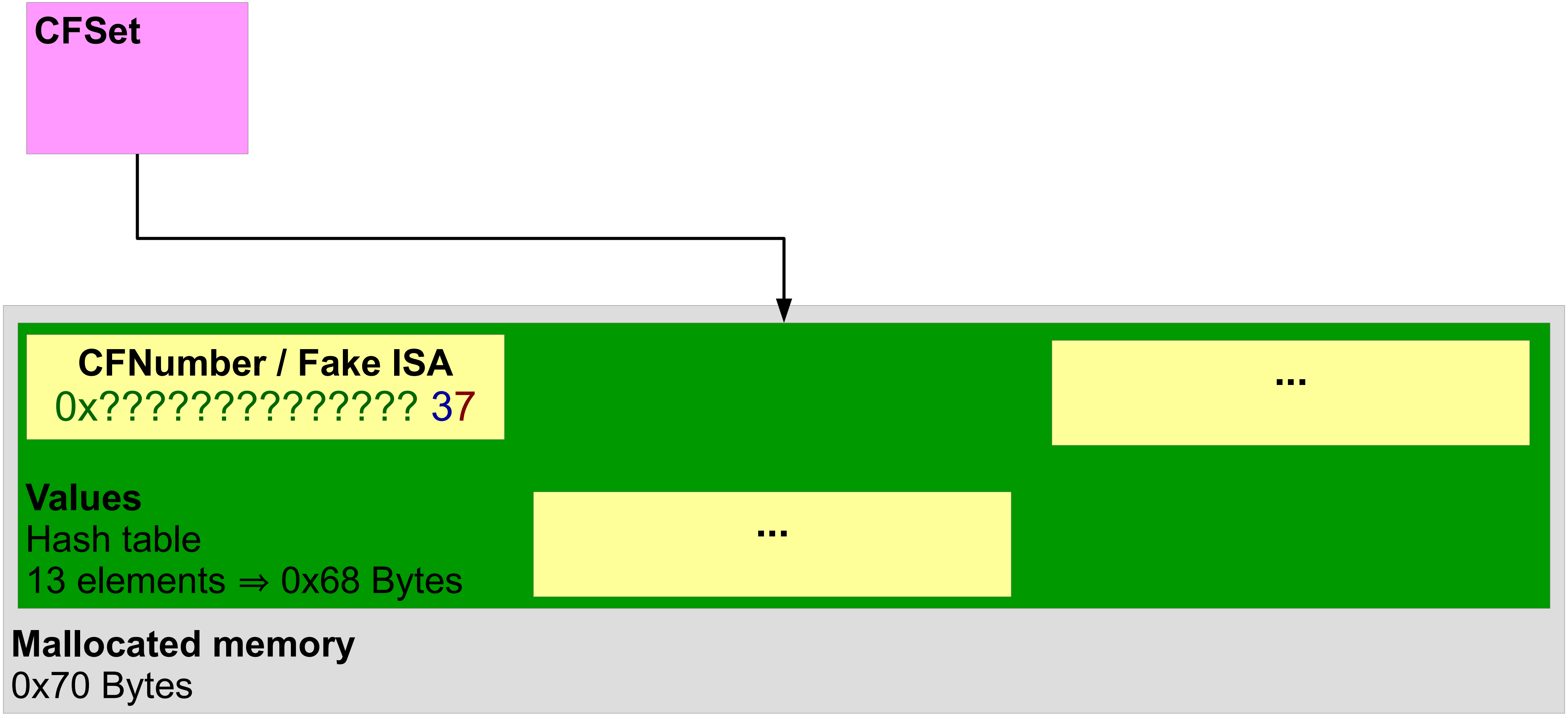
    Modern Objective-C Exploitation Techniques

    http://www.phrack.org/issues/69/9.html
- **We need to control the first `QWORD` of the allocation**
  - The ISA pointer
  - Obviously not possible with Objective-C / CoreFoundation objects

    Because they start with there own ISA pointer
- **`CFArrays` containers are inline allocated**
  - They are immutable
- **`CFSet` hash tables are not!**
  - We can forge our object in a `CFSet` hash table with `CFNumbers`!
  - Application object is 0x70 bytes wide
  - A `CFSet` with 7 to 11 elements will have a hash table of 13 elements = 0x68 bytes

    See `__CFBasicHashTableCapacities` and `__CFBasicHashTableSizes` in `CFBasicHash.c`

Battle plan


CFSet
CFNumber / Fake ISA
0x?????????????? 37
...
Values
Hash table
13 elements ⇒ 0x68 Bytes
...
Mallocated memory
0x70 Bytes

Plan

- **Massage the heap**
  - Create holes for the application object
  - Create holes just a little smaller to make sure our holes won't be "stolen" by other allocations
- **Create our application**
  - With some luck, it'll be located in one of our holes
- **Trigger the vulnerability**
- **Free the application object**
- **Reuse the allocation with our forged `CFSet` hash table**

Problem…

- **CoreFoundation uses the default heap**
  - As all the “normal” C/C++ allocations (malloc/new)
- **The default heap uses separate magazine per core**
  - Optimize the processor caches accesses
  - Reduce the risk of concurrent access (less locks)
- **This is problematic:**
  - If our object was allocated by a core, the reallocation must be done by the same core
  - We need to massage all the magazines
- **More info about the default heap in “Heapple Pie: The macOS/iOS default heap”**
  - Presented in 2018 at Sthack

    https://www.sthack.fr/
  - Slides are available

    https://www.synacktiv.com/ressources/Sthack_2018_Heapple_Pie.pdf

Massaging the heap - first idea

- **`CFSet` saved the day already twice… why not a third one?**
- **Let's forge a serialized `CFSet` with duplicated keys**
- **When a `CFSet` is deserialized**
  - all the objects are first deserialized
  - then they are put in a new `CFSet`
- **Only one of the duplicated key should be kept**
  - Others should be freed
  - This should punch holes in the heap!
- **But duplicated objects are actually never freed…**
  - `CFSet` is supposed to be correctly serialized
  - Reference counting is therefore disabled during `CFSet` creation

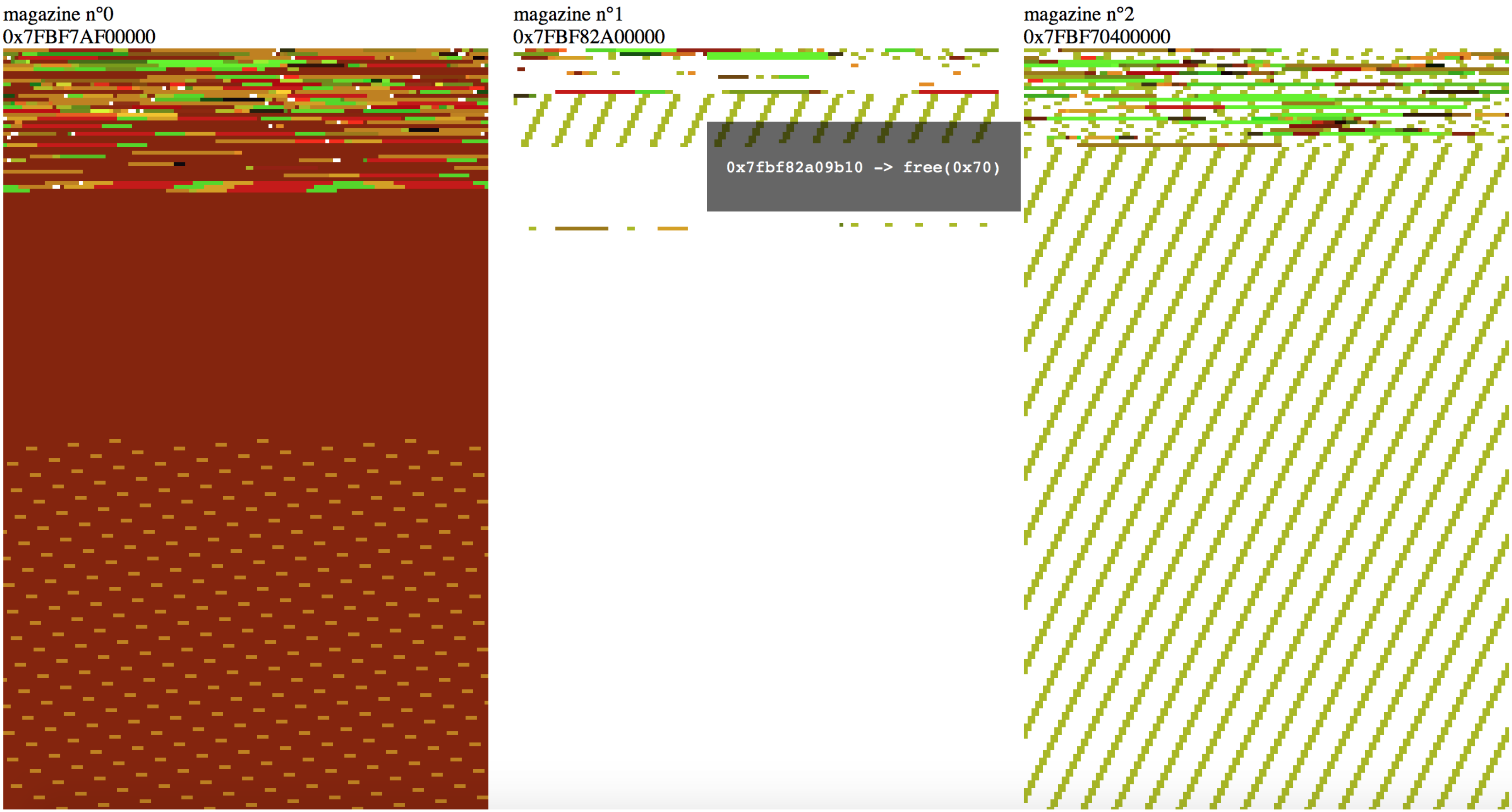
    `CFBasicHashSuppressRC` called in `__CFSetCreateTransfer`
  - It saves a `CFRetain` during the insertion and a `CFRelease` after
- **Memory leak!**

Massaging the heap - second idea

- **Just use properties**
  - Create a lot of properties
  - Free some to create holes
- **Create some applications between allocations**
  - Seems to help WindowServer to switch from a core to another one…
  - Maybe because of asynchronous operations…

    It just works
  - Helps to massage all the magazines
- **This is where a heap viewer comes in handy :)**

Heap viewer


magazine n°0
0x7FBF7AF00000
magazine n°1
0x7FBF82A00000
magazine n°2
0x7FBF70400000
0x7fbf82a09b10 -> free(0x70)

Step 4

Trigger the use of the overwritten pointer
to gain arbitrary code execution

Getting code exec

- **Triggering the use of our overwritten pointer:**
  - We just have to redefine / delete the property associated with our overwritten `CFSet`
- **This will call `CFRelease` with our overwritten pointer**
- **But how do we get arbitrary code exec?**
  - By forging a fake Objective-C object and a fake Objective-C Class
  - Multiple dereferences are involved before getting RIP

    Object → ISA pointer → Class → cache → function pointer
- **How are we going to bypass ASLR?**
  - For code: easy, system libs are loaded at the same address for all processes thanks to the shared cache

    Not exactly true anymore on iOS12 but that’s an other story
  - For data?

Bypassing ASLR

- **We could leak the reused object address**
  - Like ret2 Systems did
- **But the object is quite small…**
  - Hard to fit everything in it…
- **macOS/iOS ASLR is known to be weak**
  - phoenhex.re exploit for CVE-2017-2536 – 32GB of spray
    - WebKit heap – possible due to page compression
  - Brandon Azad exploit for CVE-2018-4331 – 4 GB of spray
    - mach_vm_map – via libxpc
  - @S0rryMybad exploit for CVE-2019-6225 – unknown size
    - Kernel heap
- **Not that uncommon to hardcode addresses…**

How does the ASLR work?

- **Default heap is randomized in userland**
  - See `mvm_aslr_init` and `mvm_allocate_pages_securely` in `libmalloc`
  - Random is provided by the kernel at startup
    - Via applev special environment variables
- **Heap start address = min stack addr - random - 256MiB**
  - x86_64: 16bits of random, 8MiB step ⇒ 512GiB spray needed
  - aarch64: 7 bits, 32MiB step ⇒ 4GiB spray needed
  - Others: 3bits, 8MiB step ⇒ 64MiB spray needed
    - But other mitigations are used for those architectures
    - First n blocks are dismissed for each regions
- **Not practical…**

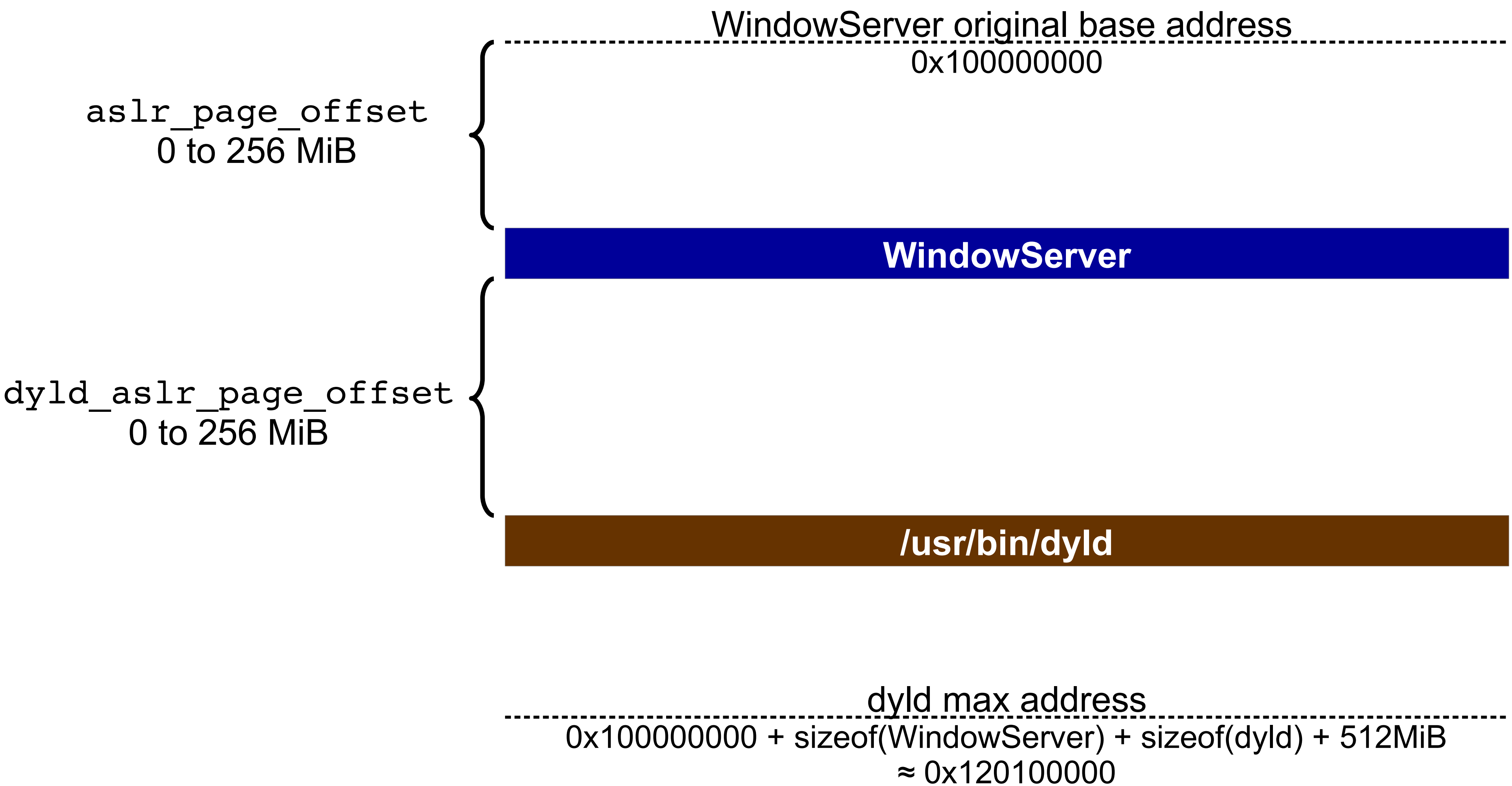
How does the ASLR work?

- **The main executable base address is randomized**
  - If it has a `__PAGEZERO` segment
    - vmaddr = filesize = 0
    - initprot = maxprot = VM_PROT_NONE
    - vmsize ≠ 0
    - Name isn't actually important
  - A random slide, `aslr_page_offset`, is added to the original base address
  - Different for each architecture
    - 0 to 80/20 MiB on aarch64 16K/4K
    - 0 to 256 MiB on x86_64
    - 0 to 1 MiB on x86
- **The same slide is also directly used to randomize the stack base address**
  - leak of the stack ⇔ leak of the main executable address
- **See `load_machfile` in `bsd/kern/mach_loader.c` for more information**

How does the ASLR work?

- **`/usr/bin/dyld` is loaded after the main executable**
  - If it doesn't have a base address
    - Always the case on macOS and iOS
    - Otherwise it is loaded like the main executable
  - A new random slide, `dyld_aslr_page_offset`, is added to the end of the executable address to get `dyld` base address
  - Different for each architecture
    - 0 to 4 MiB on aarch64
    - 0 to 256 MiB on x86_64
    - 0 to 1 MiB on x86
- **Again, see `load_machfile` in `bsd/kern/mach_loader.c` for more details**

Battle plan


WindowServer original base address
0x100000000
aslr_page_offset
0 to 256 MiB
WindowServer
dyld_aslr_page_offset
0 to 256 MiB
/usr/bin/dyld
dyld max address
0x100000000 + sizeof(WindowServer) + sizeof(dyld) + 512MiB
≈ 0x120100000

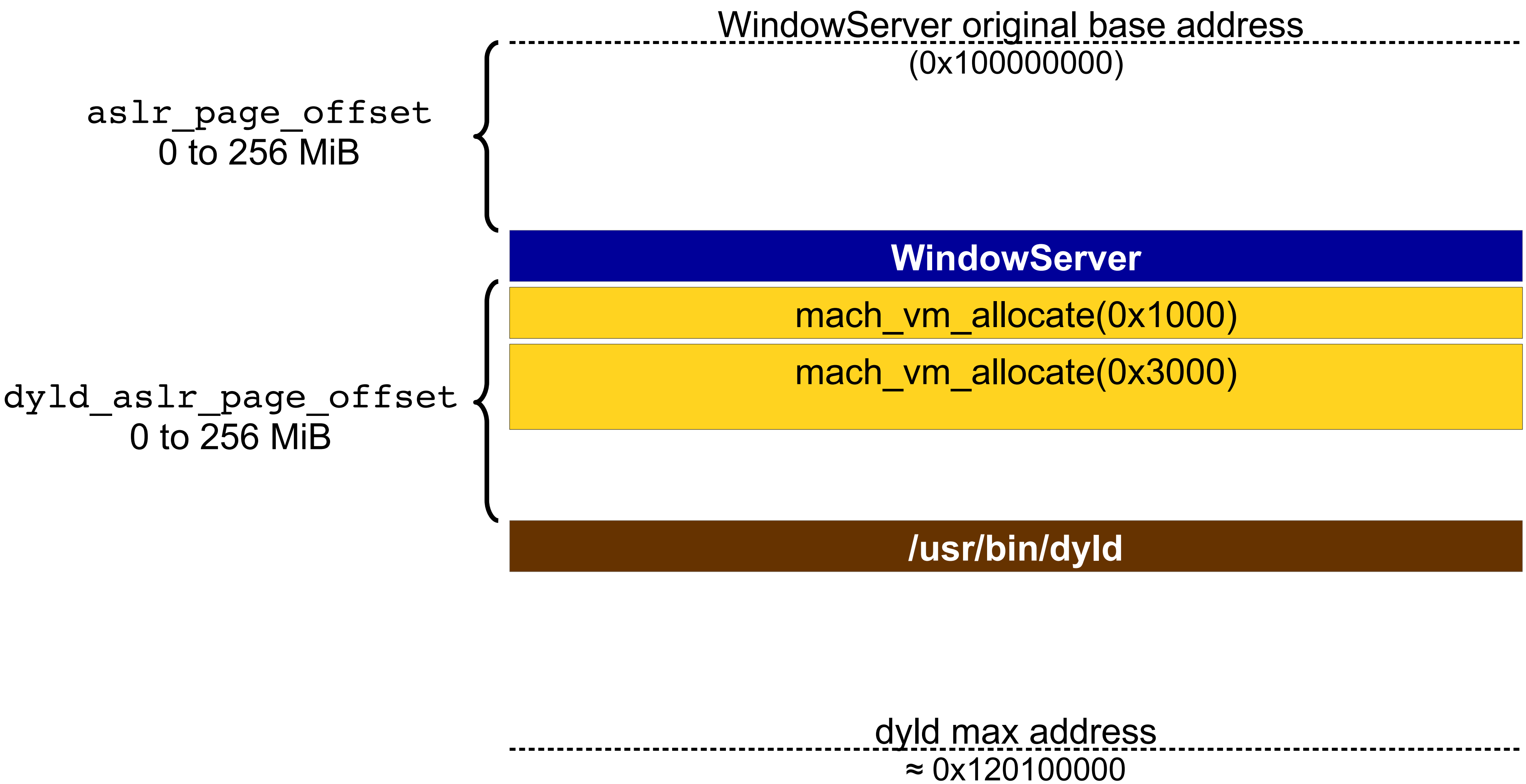
How does the ASLR work?

- **Pages allocated without a specific address are allocated just after the main executable**
  - `vm_map_raise_min_offset` is used to block all allocations before the `__PAGEZERO` segment when it is “loaded”

    see `load_segment` in `bsd/kern/mach_loader.c`
- **Unless `posix_spawn` is used with the undocumented flag `_POSIX_SPAWN_HIGH_BITS_ASLR`...**
  - Only valid on x86_64
  - `start_address = (random() & 0xFF) << 27`
  - Only impact `vm_map_enter`
  - Used by WebKit XPC services

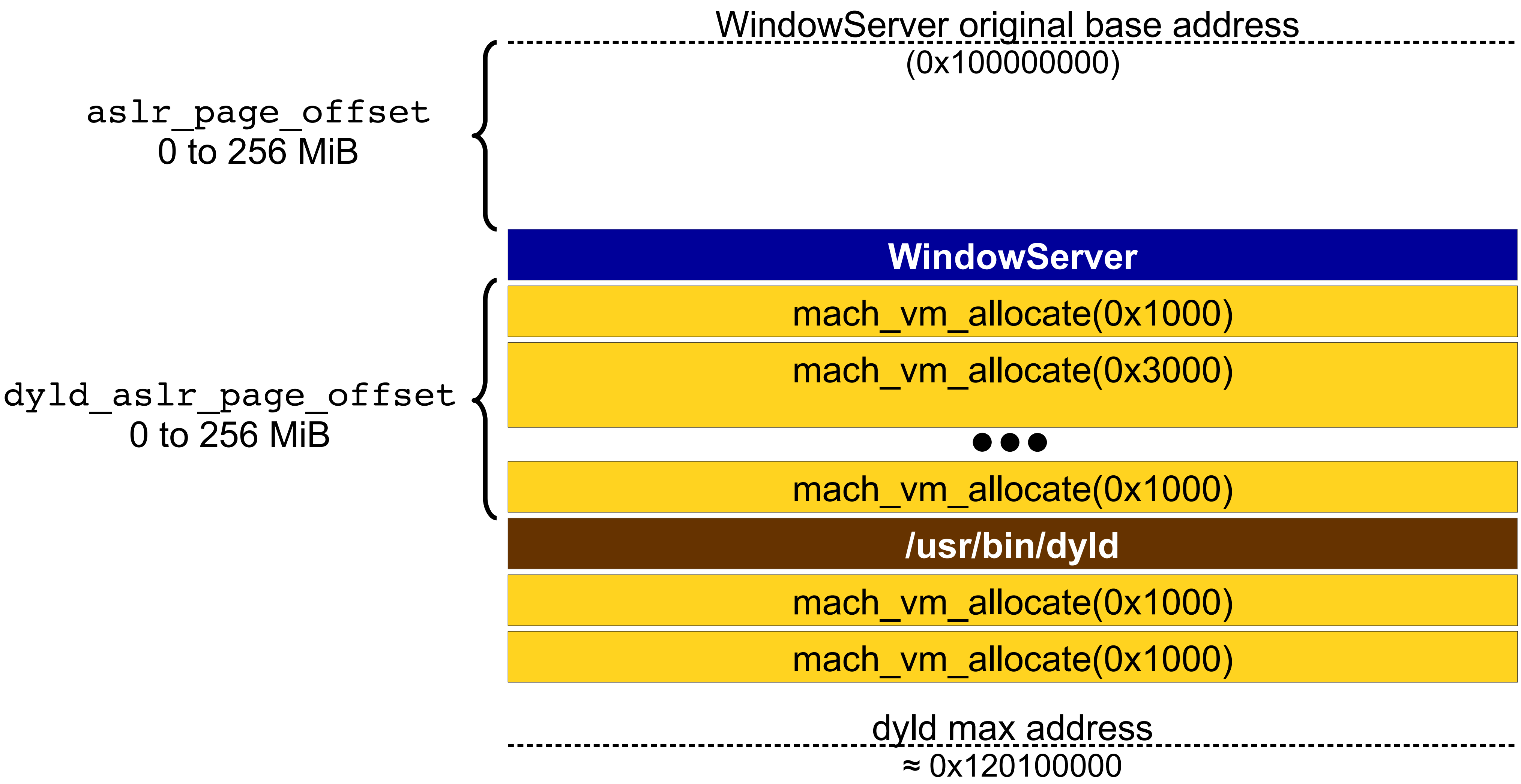
    `com.apple.WebKit.WebContent[.Development]` and `com.apple.WebKit.Plugin` activated by the undocumented `XPCService→_HighBitsASLR` key in the service `Info.plist`
- **If we allocate 512+ MiB via `mach_vm` APIs in the distant process, we completely bypass the ASLR**
  - Even 256 MiB if you are not afraid to collide with dyld

Battle plan

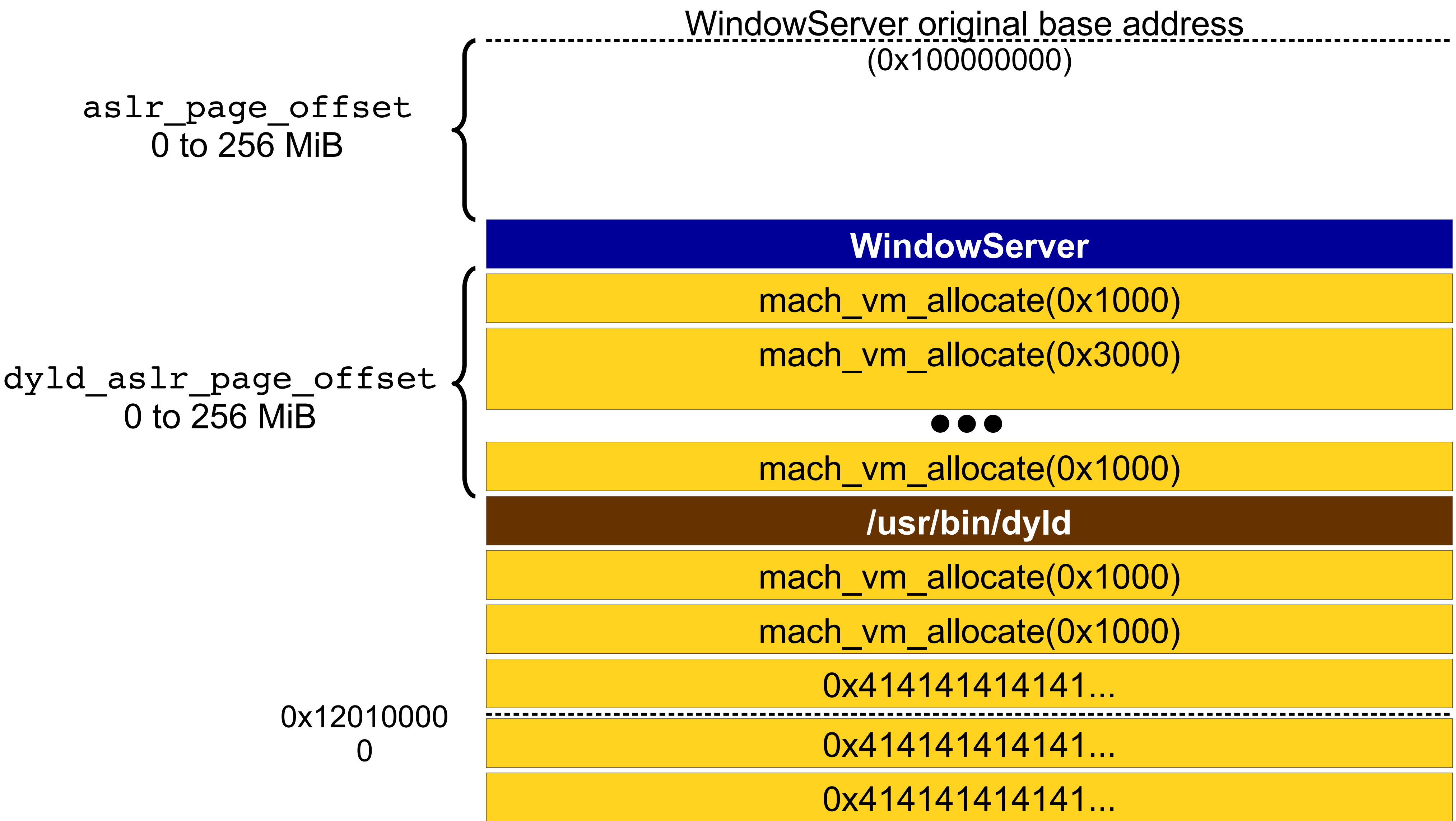


WindowServer original base address
(0x100000000)
aslr_page_offset
0 to 256 MiB
WindowServer
mach_vm_allocate(0x1000)
mach_vm_allocate(0x3000)
dyld_aslr_page_offset
0 to 256 MiB
/usr/bin/dyld
dyld max address
≈ 0x120100000

Battle plan


WindowServer original base address
(0x100000000)
aslr_page_offset
0 to 256 MiB
WindowServer
dyld_aslr_page_offset
0 to 256 MiB
mach_vm_allocate(0x1000)
mach_vm_allocate(0x3000)
mach_vm_allocate(0x1000)
/usr/bin/dyld
mach_vm_allocate(0x1000)
mach_vm_allocate(0x1000)
dyld max address
≈ 0x120100000

Battle plan



WindowServer original base address
(0x100000000)
aslr_page_offset
0 to 256 MiB
WindowServer
mach_vm_allocate(0x1000)
mach_vm_allocate(0x3000)
dyld_aslr_page_offset
0 to 256 MiB
mach_vm_allocate(0x1000)
/usr/bin/dyld
mach_vm_allocate(0x1000)
mach_vm_allocate(0x1000)
0x414141414141...
0x120100000
0x414141414141...
0x414141414141...

mach_msg

- **WindowServer is an old service**
  - Not a fancy XPC or NSXPC service
  - Good old MIG server
- **Unbonded arrays are passed as out-of-line descriptors**
  - Size must fit in a DWORD (< 4GiB)
  - No other restrictions
  - Arrays are freed just after the execution of the MIG handler
- **For large arrays (≳ 2 pages), XNU use copy-on-write mechanisms**
  - Almost no physical memory is used

    Except for the page table
  - First free large-enough pages of the map are used (see `vm_map_copyout`)

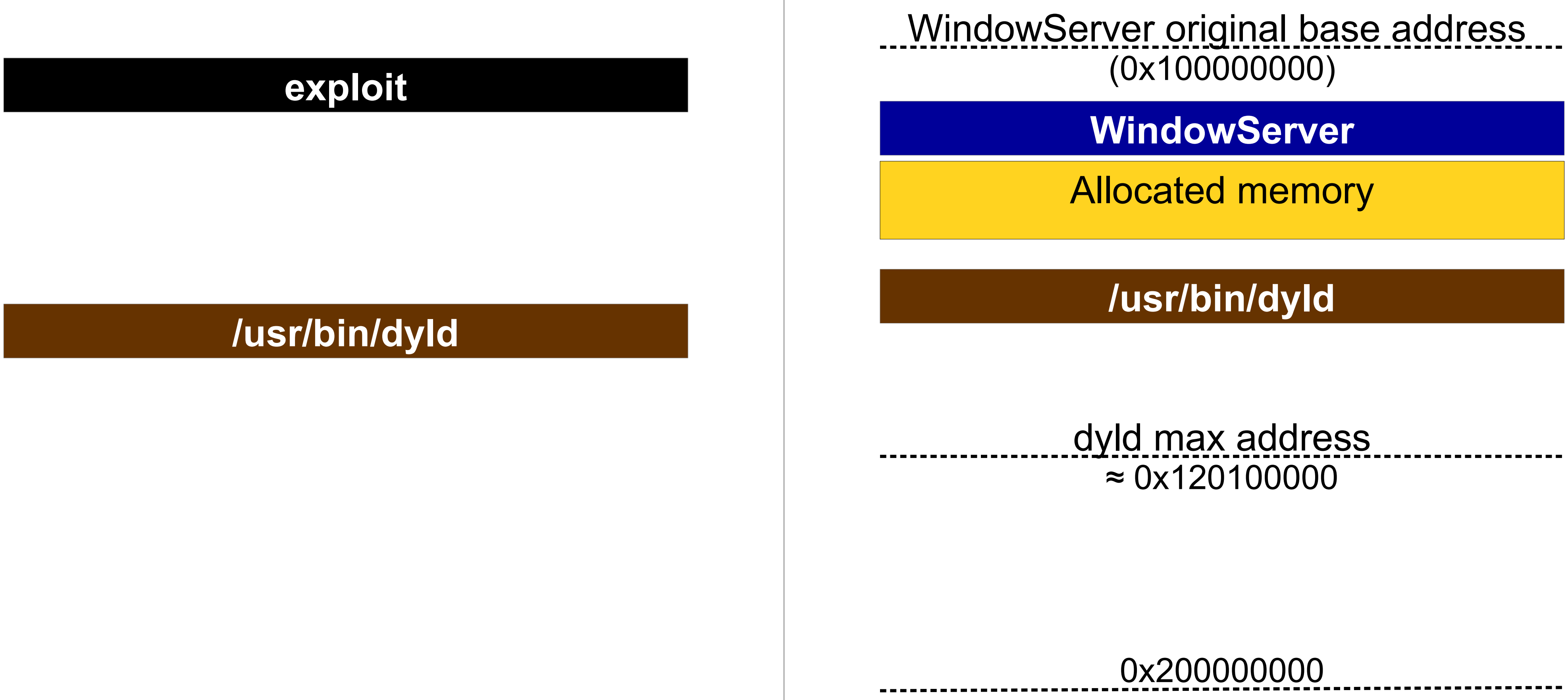
    Those just after the executable :)

    Even if `_POSIX_SPAWN_HIGH_BITS_ASLR` is used
  - Very fast, even more if the memory is deallocated from the sender (no COW needed)

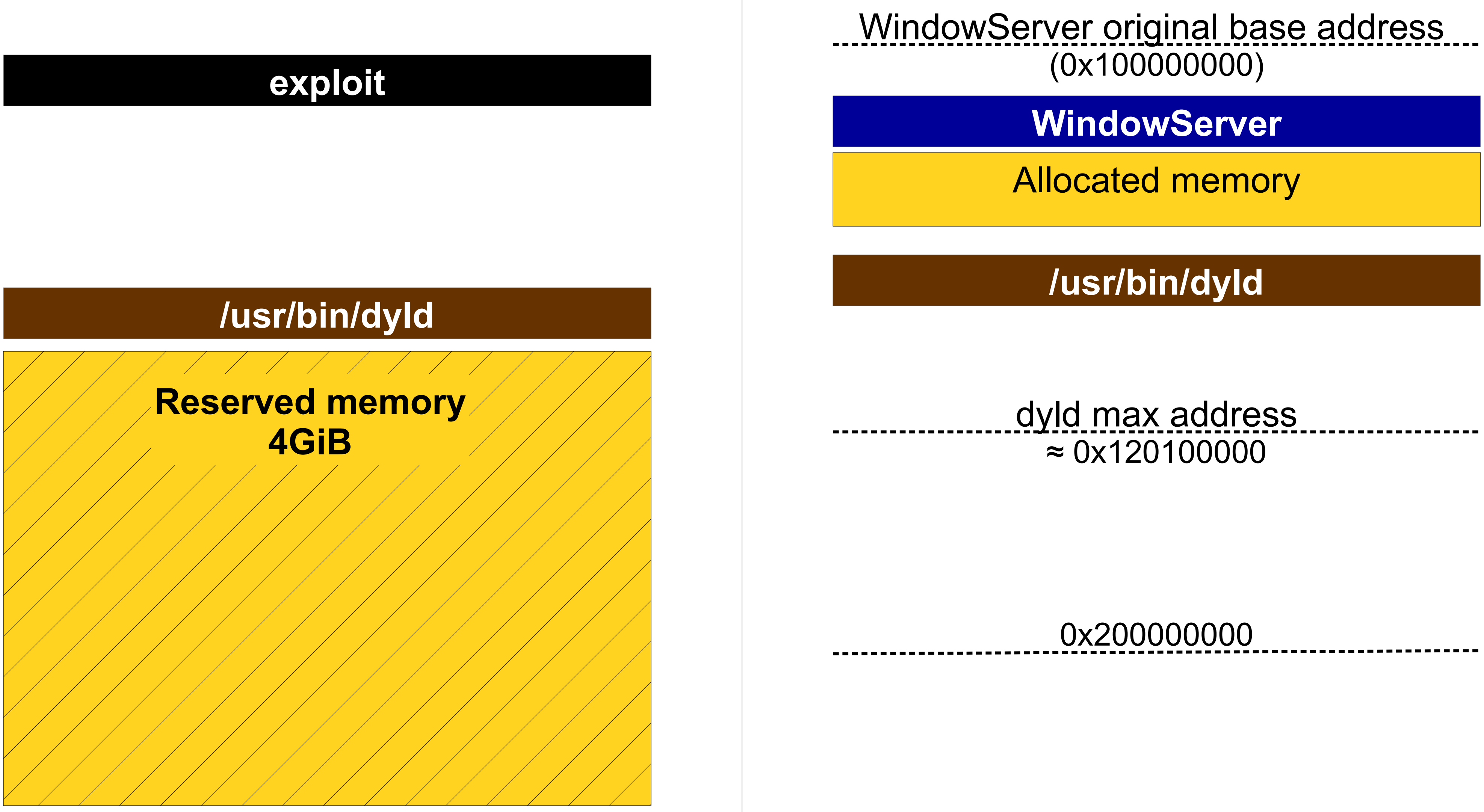
Strategy

- **Create a LOT of contiguous pages starting with a fake Obj-C classes**
  - Reserve 4GiB of **virtual** memory
  - Allocate a page in it and put our fake Obj-C class with our payload
  - Remap the page 0xFFFFF times to create a 4GiB contiguous buffer
  - This will use very few physical pages
    - One for the payload, few others for the page table
- **Call `SetConnectionProperty` to replace the overwritten property with the 4 GiB buffer**
  - The buffer will be copied in WindowServer
  - One of the copies of our fake Obj-C is now guaranteed to be located at 0x200000000
- **Our fake class is used during the overwritten `CFSet` destruction**
  - Our payload is executed…
- **The service automatically free the sent buffer**

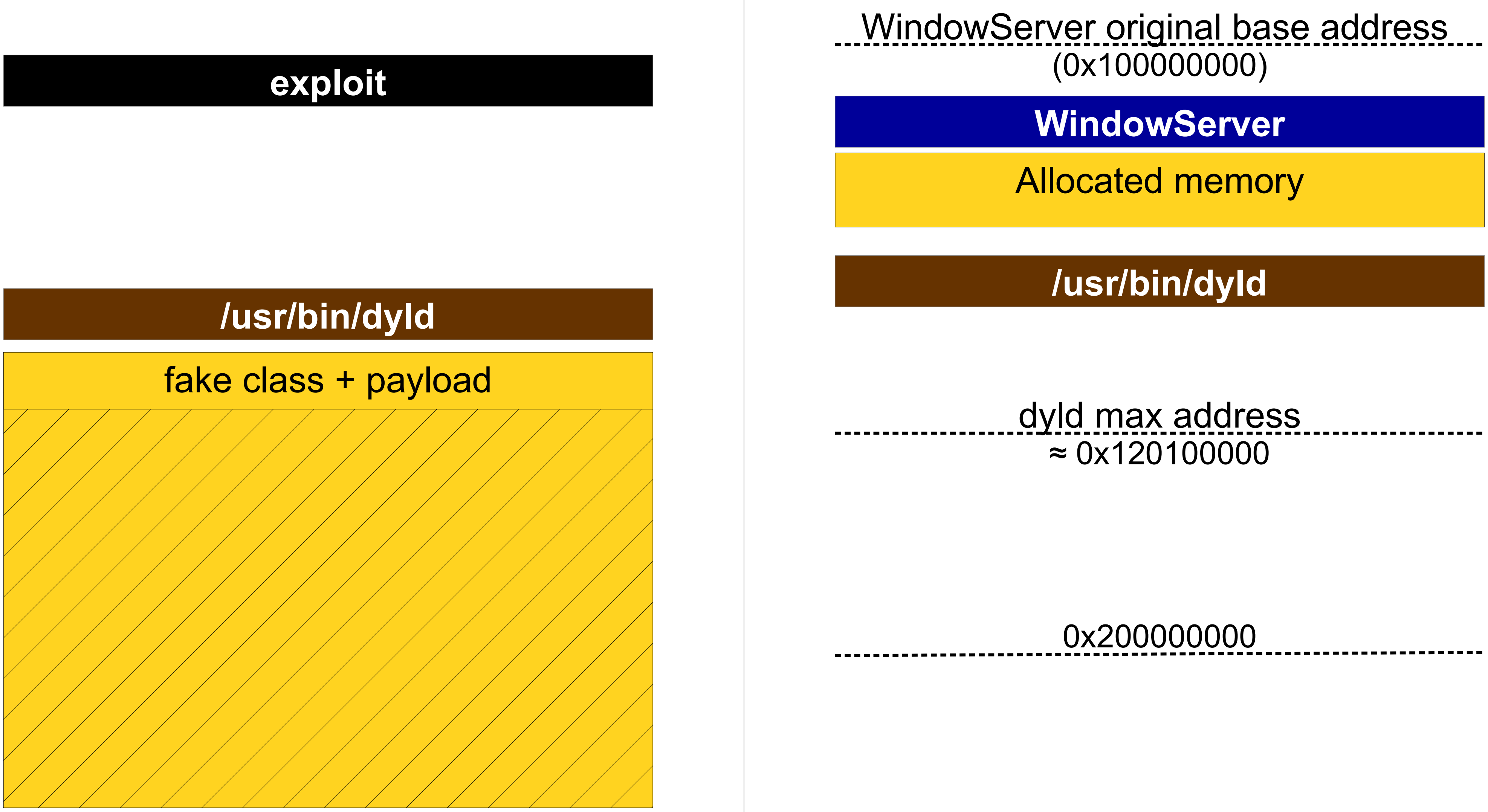
Battle plan


exploit
/usr/bin/dyld
WindowServer original base address
(0x100000000)
WindowServer
Allocated memory
/usr/bin/dyld
dyld max address
≈ 0x120100000
0x200000000

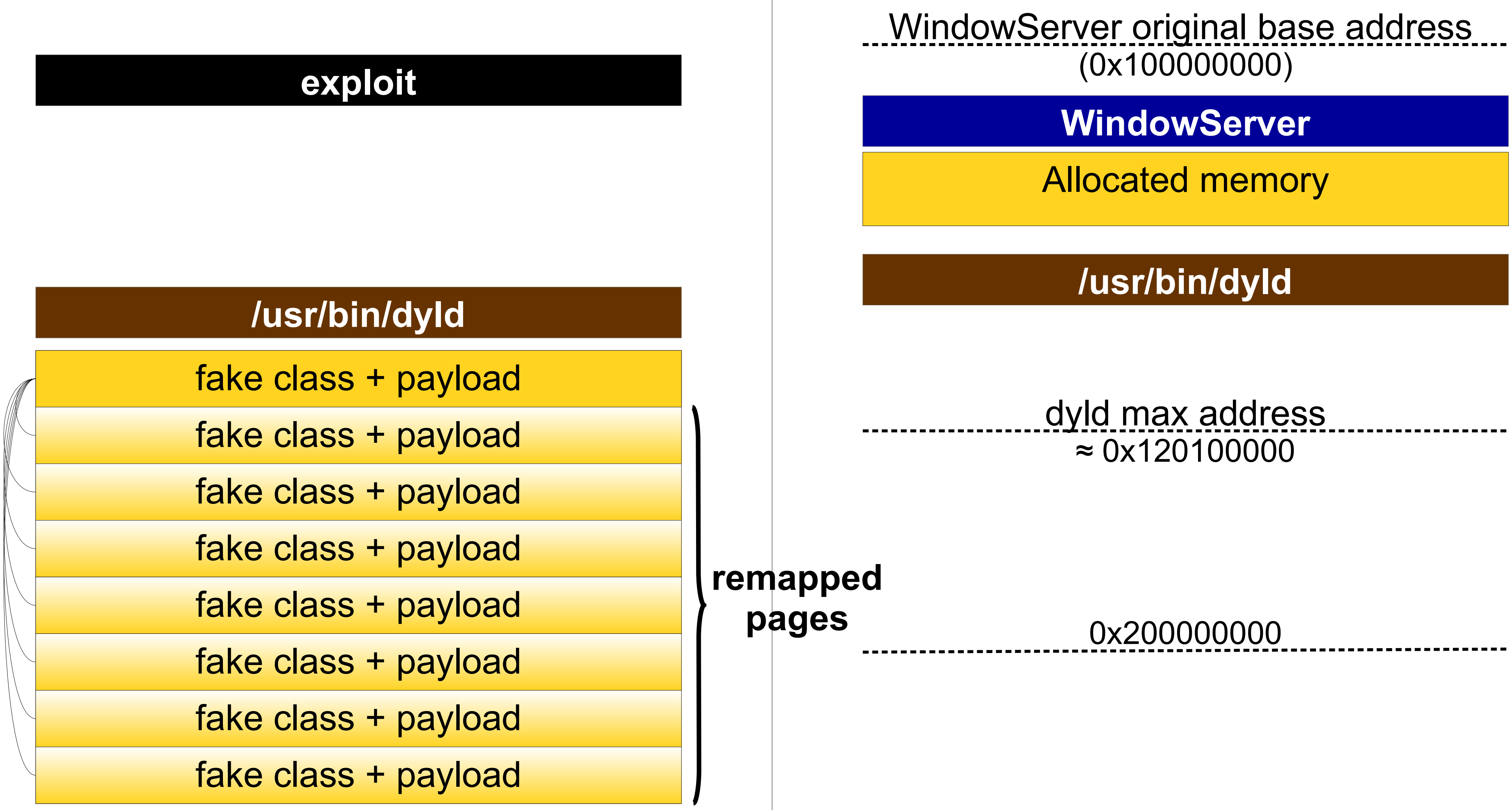
Battle plan


exploit
/usr/bin/dyld
Reserved memory
4GiB
WindowServer original base address
(0x100000000)
WindowServer
Allocated memory
/usr/bin/dyld
dyld max address
≈ 0x120100000
0x200000000

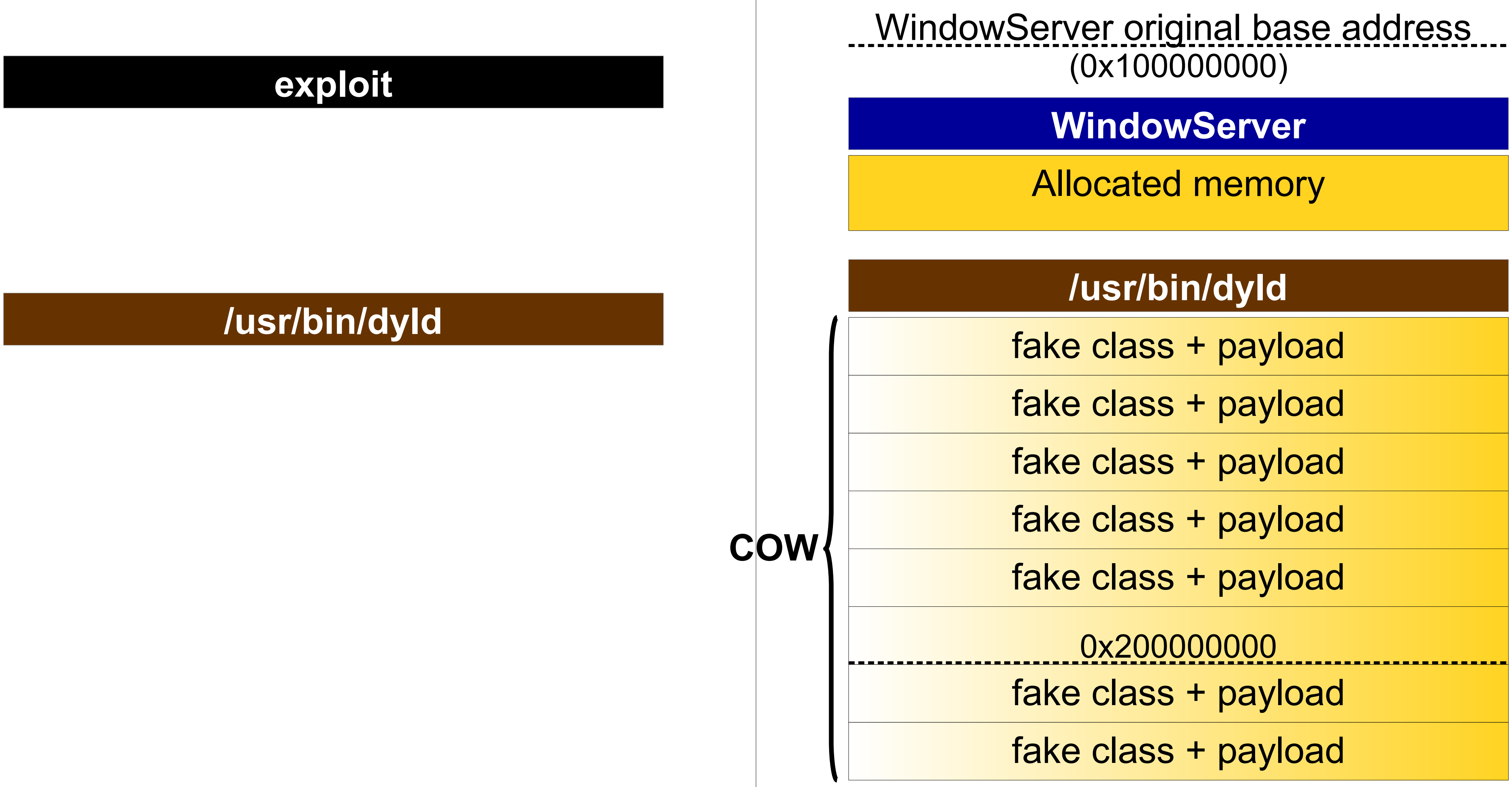
Battle plan


exploit
/usr/bin/dyld
fake class + payload
WindowServer original base address
(0x100000000)
WindowServer
Allocated memory
/usr/bin/dyld
dyld max address
≈ 0x120100000
0x200000000

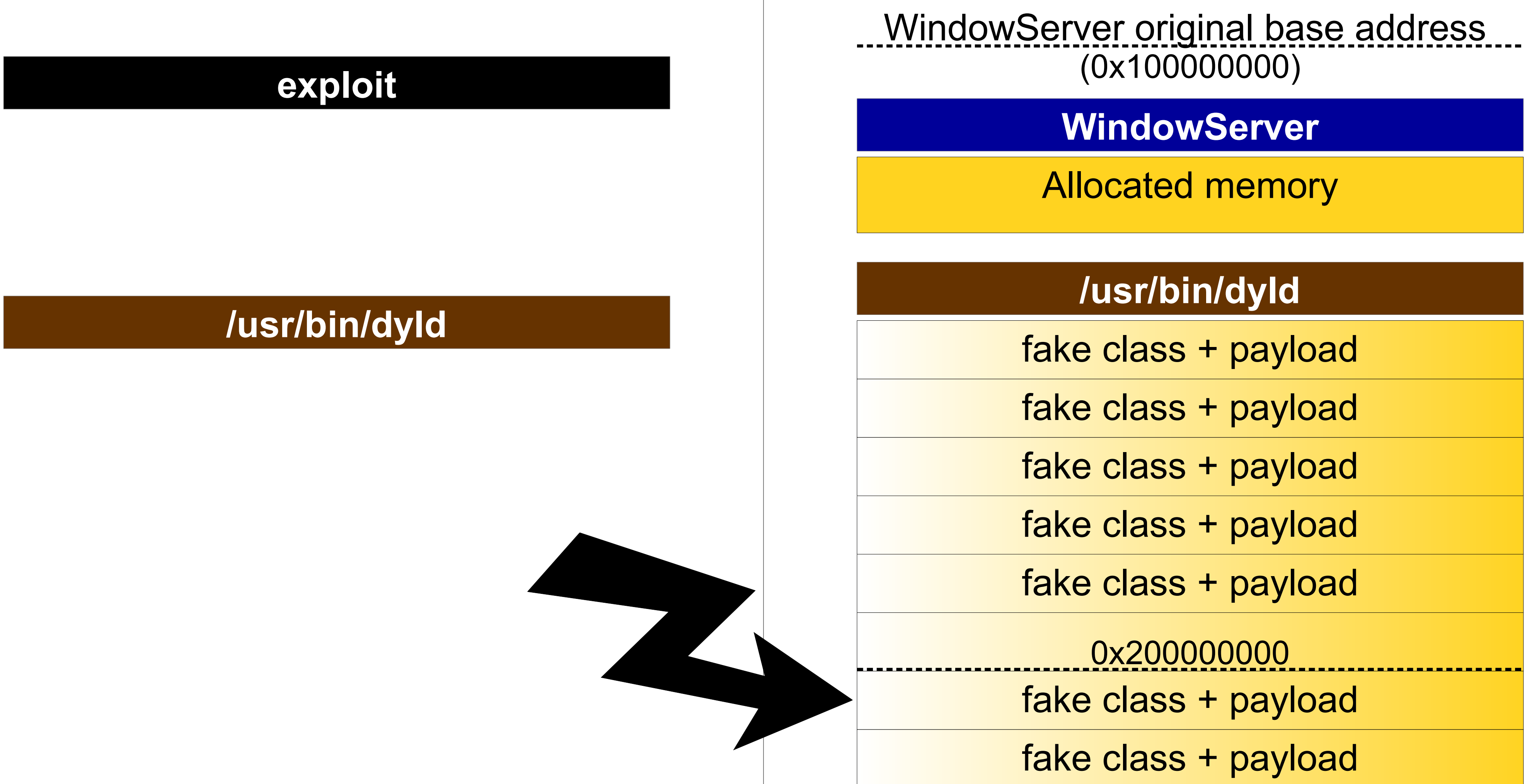
Battle plan


exploit
/usr/bin/dyld
fake class + payload
fake class + payload
fake class + payload
fake class + payload
fake class + payload
fake class + payload
fake class + payload
fake class + payload
remapped pages
WindowServer original base address
(0x100000000)
WindowServer
Allocated memory
/usr/bin/dyld
dyld max address
≈ 0x120100000
0x200000000

Battle plan


exploit
/usr/bin/dyld
WindowServer original base address
(0x100000000)
WindowServer
Allocated memory
/usr/bin/dyld
COW
fake class + payload
fake class + payload
fake class + payload
fake class + payload
fake class + payload
0x200000000
fake class + payload
fake class + payload

Battle plan


exploit
/usr/bin/dyld
WindowServer original base address
(0x100000000)
WindowServer
Allocated memory
/usr/bin/dyld
fake class + payload
fake class + payload
fake class + payload
fake class + payload
fake class + payload
0x200000000
fake class + payload
fake class + payload

Battle plan


exploit
/usr/bin/dyld
WindowServer original base address
(0x100000000)
WindowServer
Allocated memory
/usr/bin/dyld

Step 5

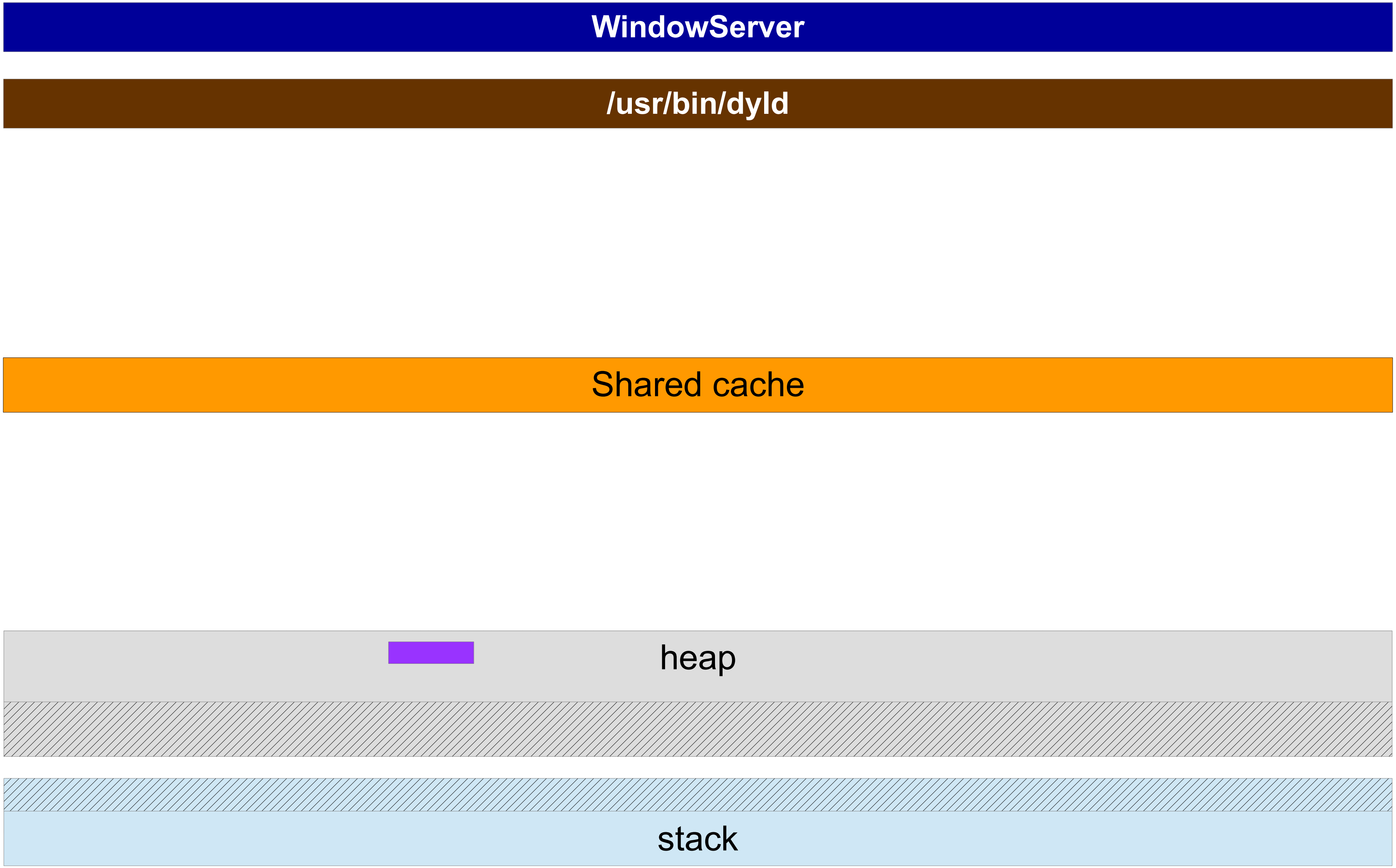
Execute our payload and ensure continuation of execution

Strategy

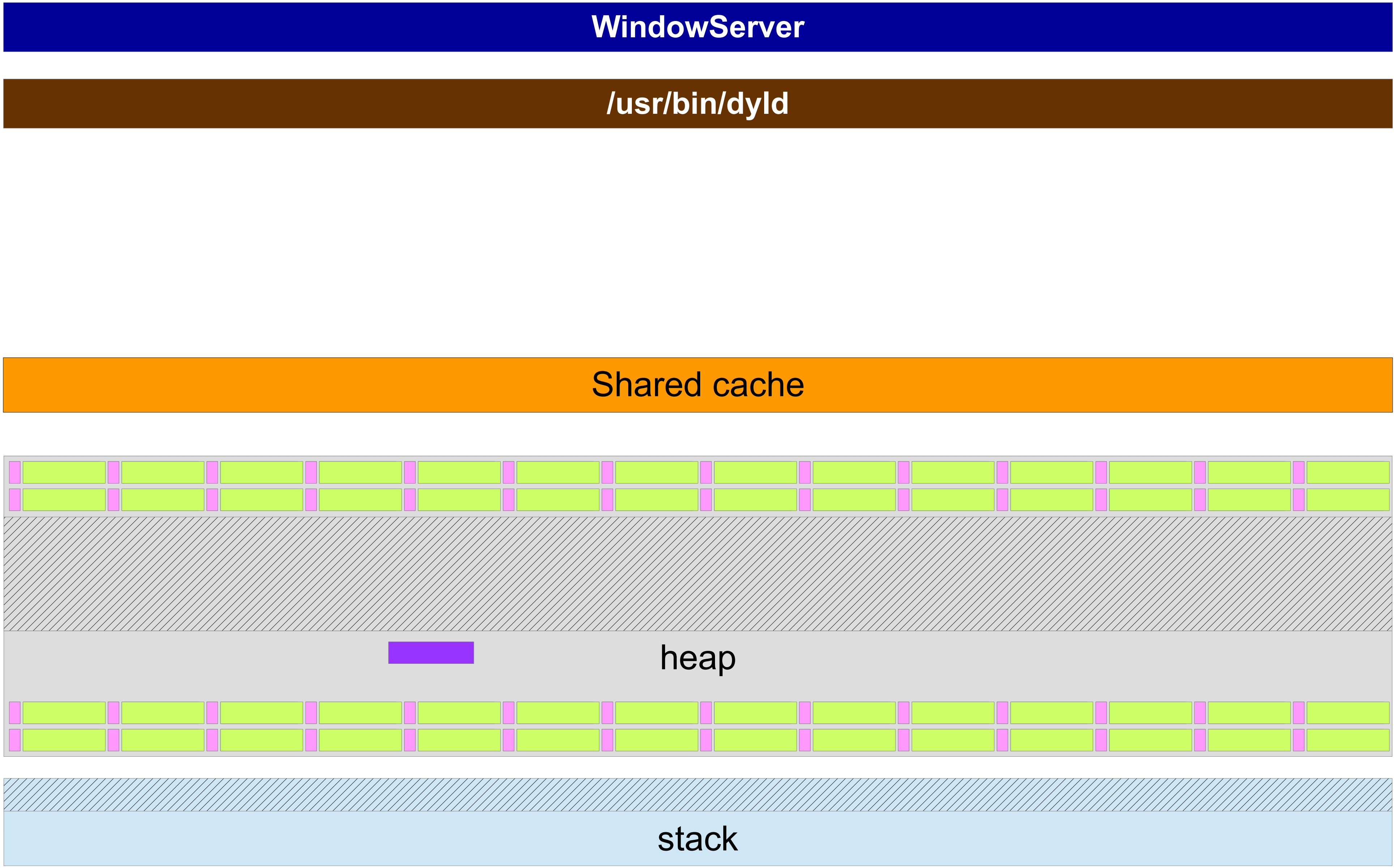
- **RDI points on our Obj-C forged object and we control RIP**
- **We could try to stack pivot and ROP**
  - That's what ret2 Systems did
  - But complicated to ensure continuation of execution
    - How to restore the original RSP value?
- **Pure JOP payload sounded like a better option…**
  - Use a JOP chain to set RDI
  - Jump on system
  - 4 gadgets (dynamically found) to get arbitrary command execution

Summary

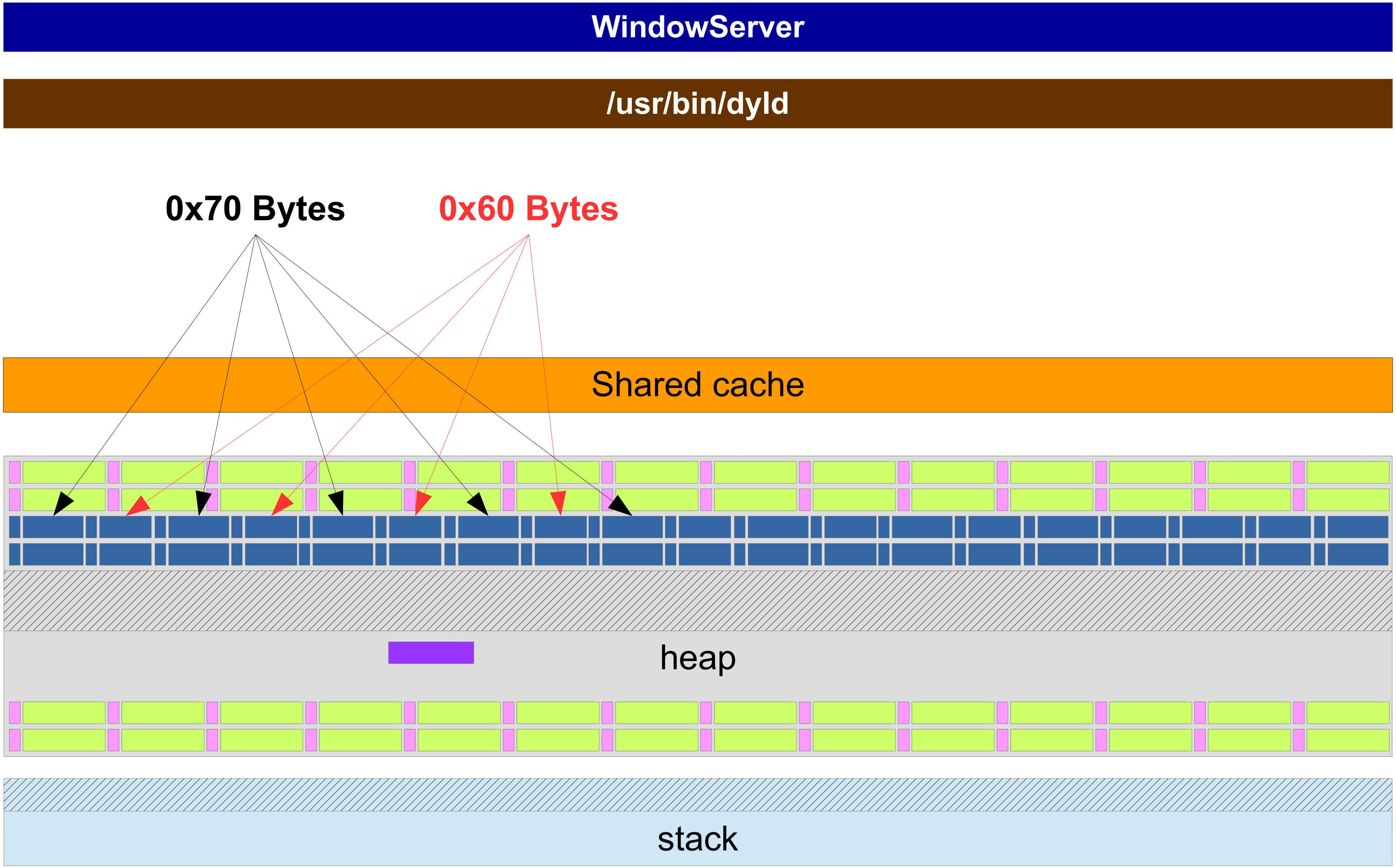
Initial state



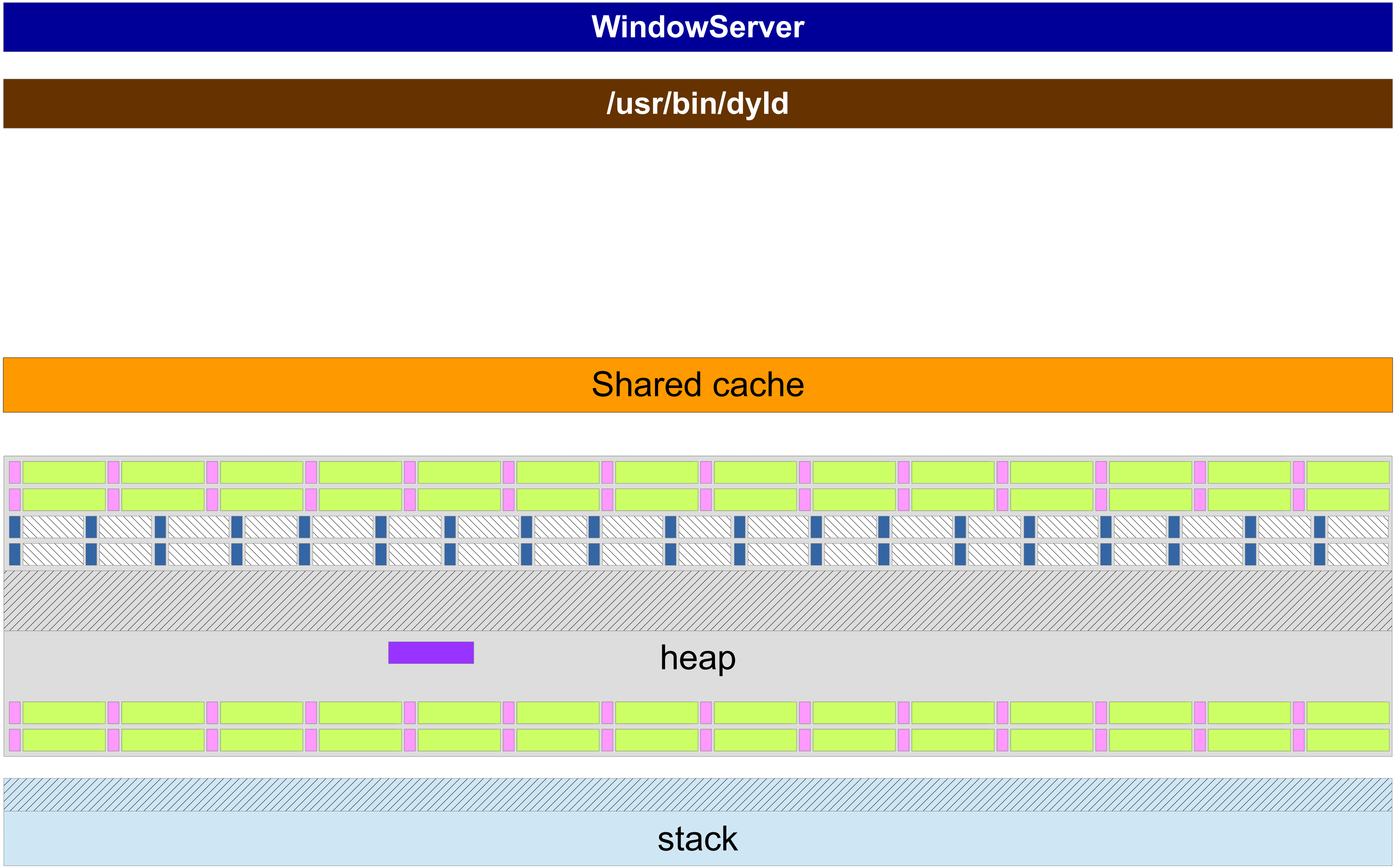
Heap spray



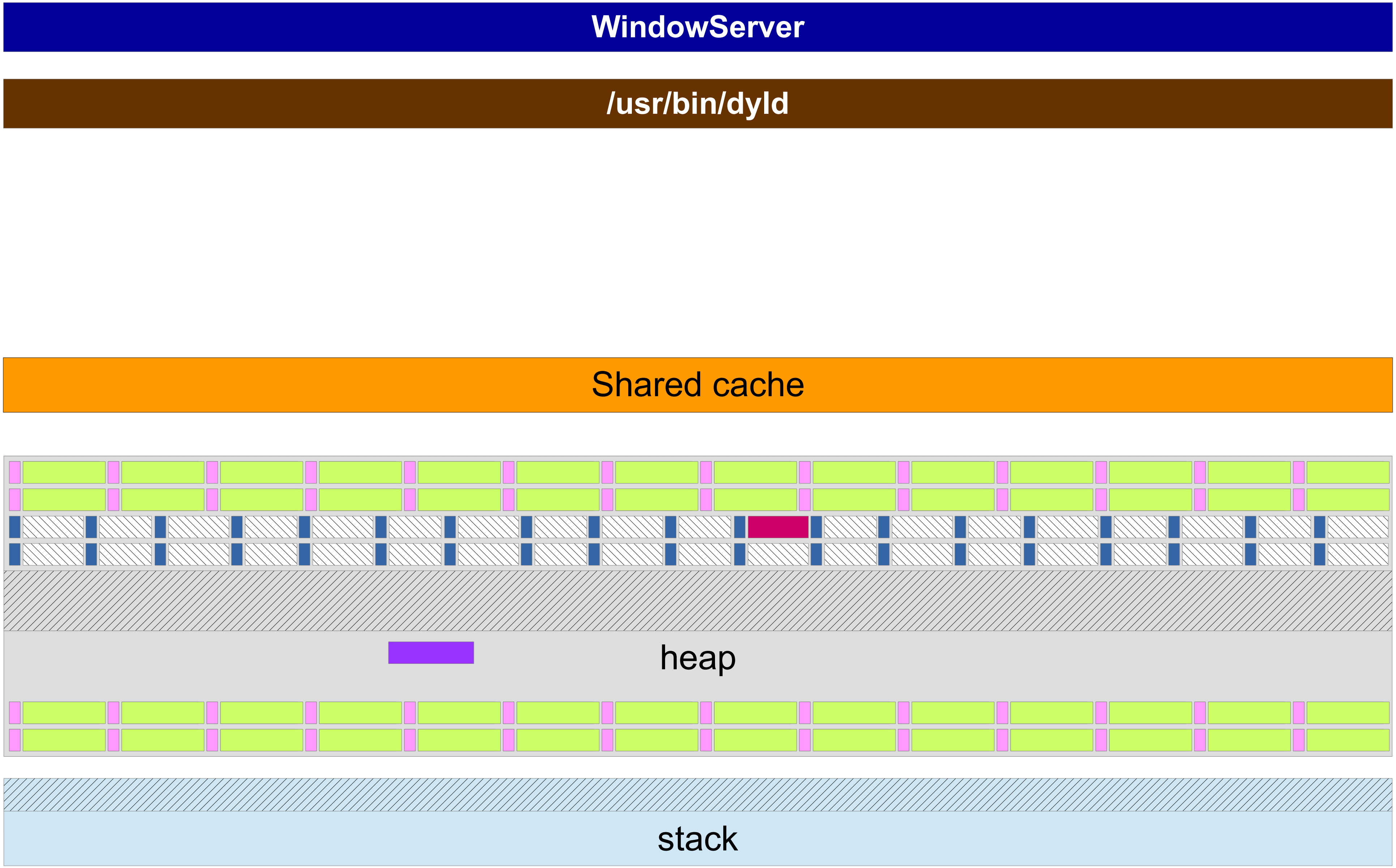
Heap Massaging 1/2



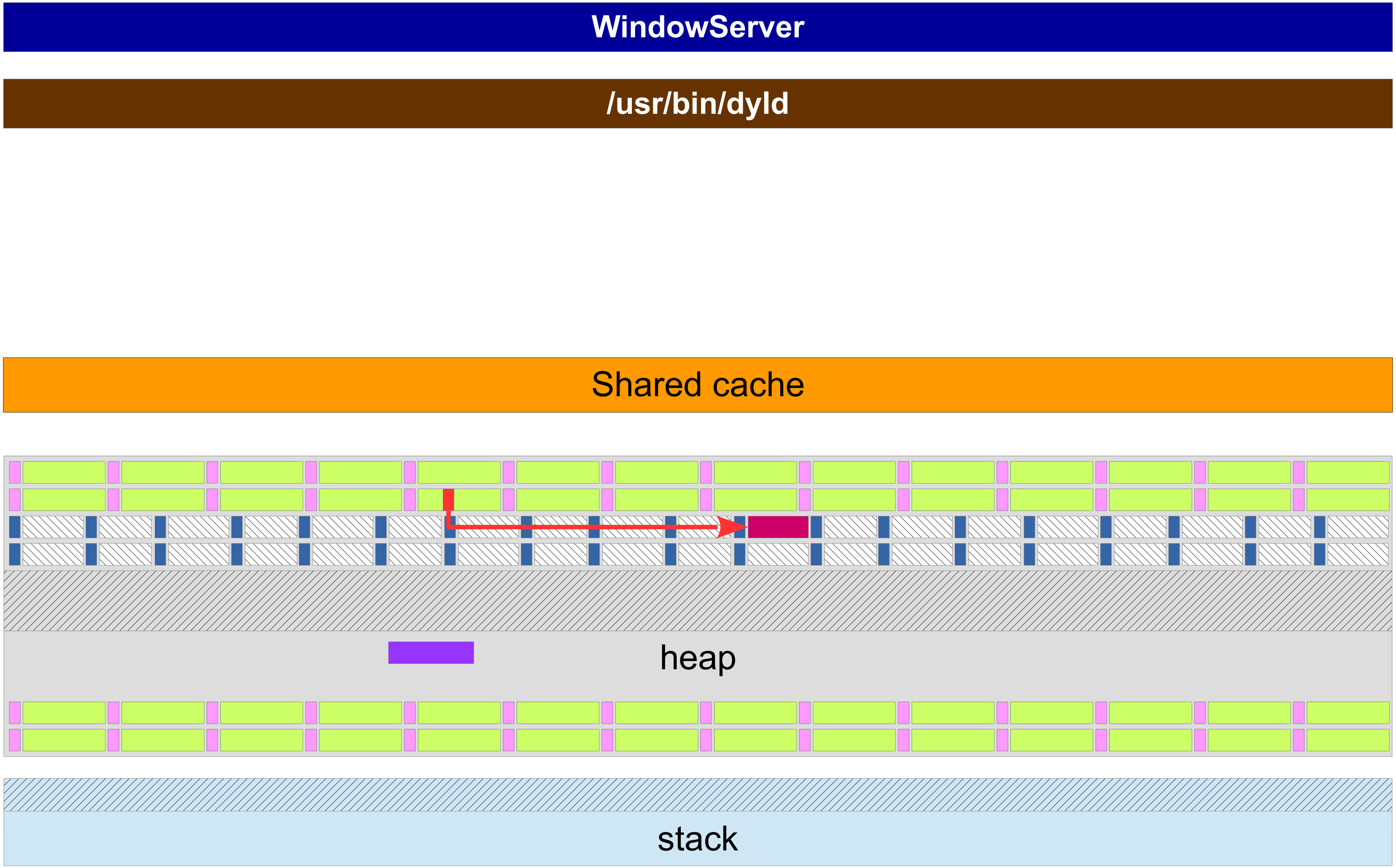
Heap Massaging 2/2



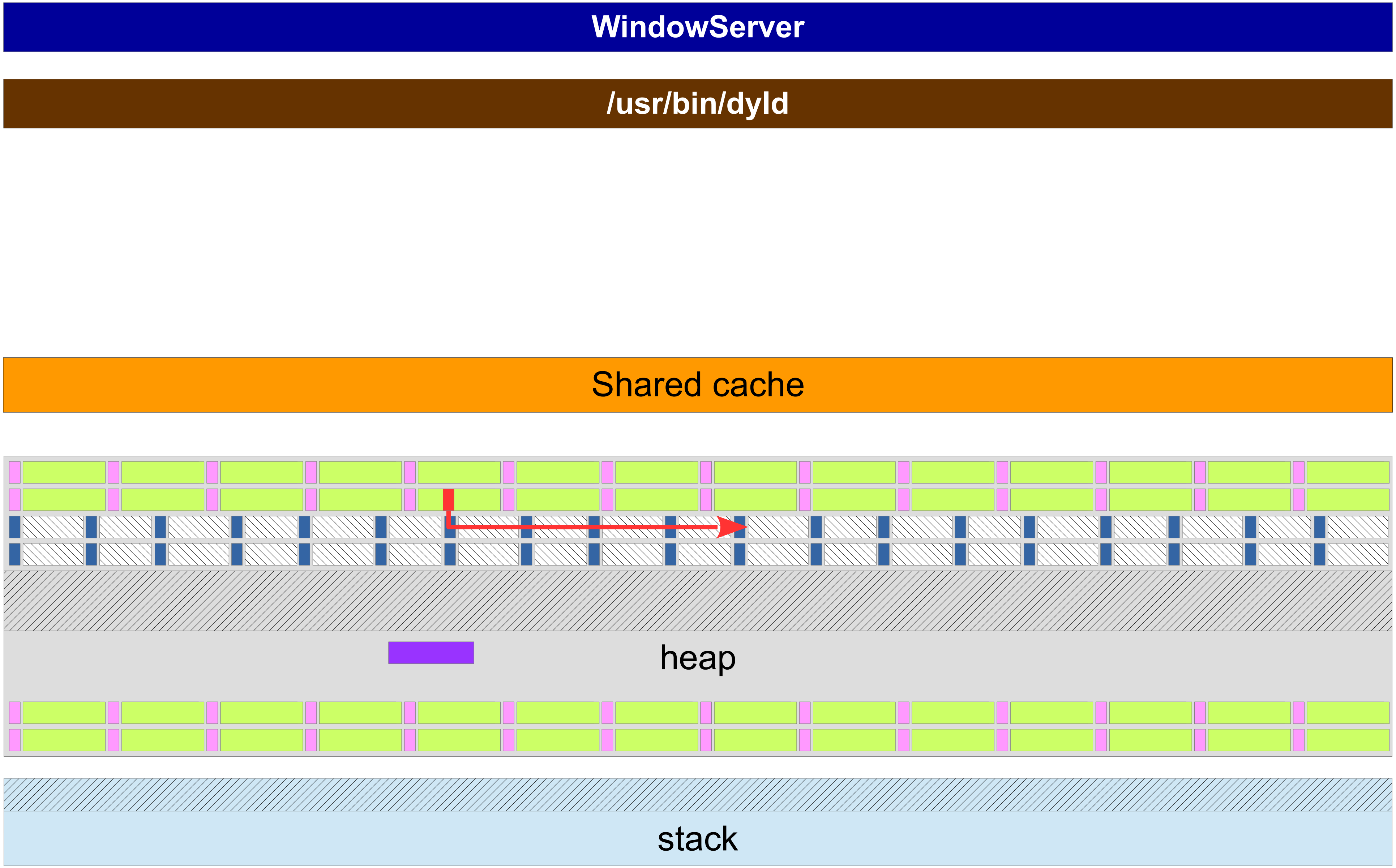
Application allocation



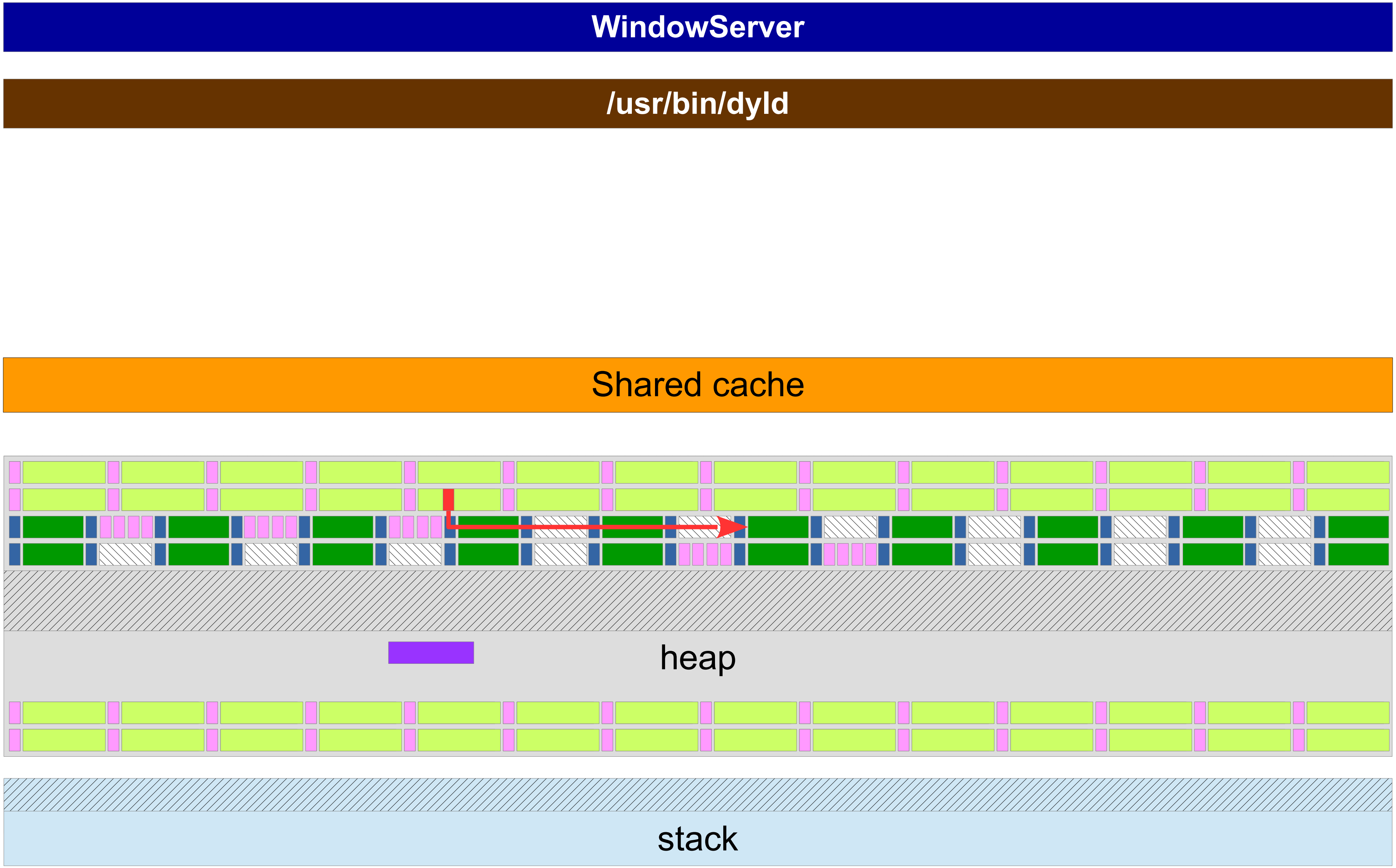
Vulnerability triggering



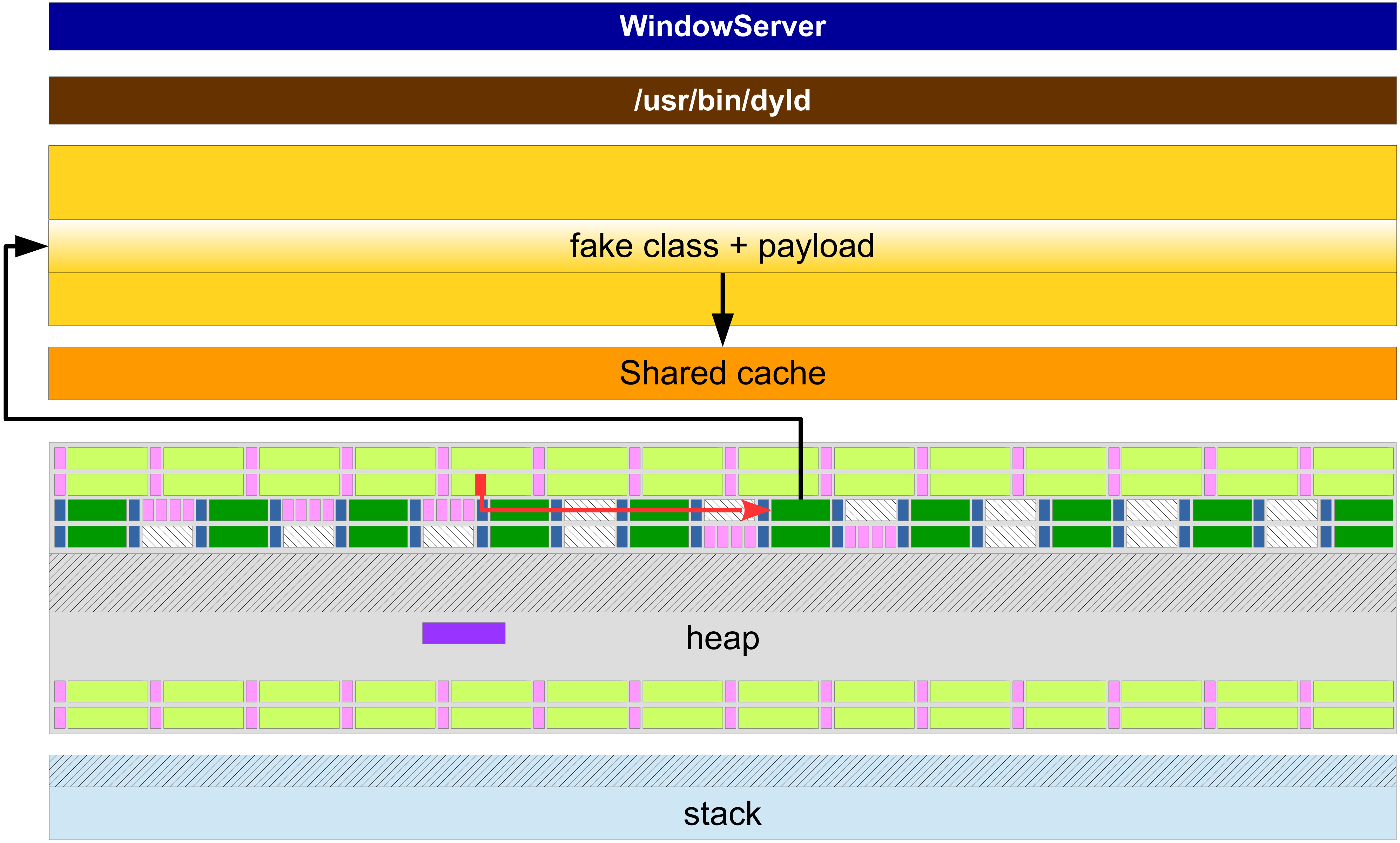
Application destruction



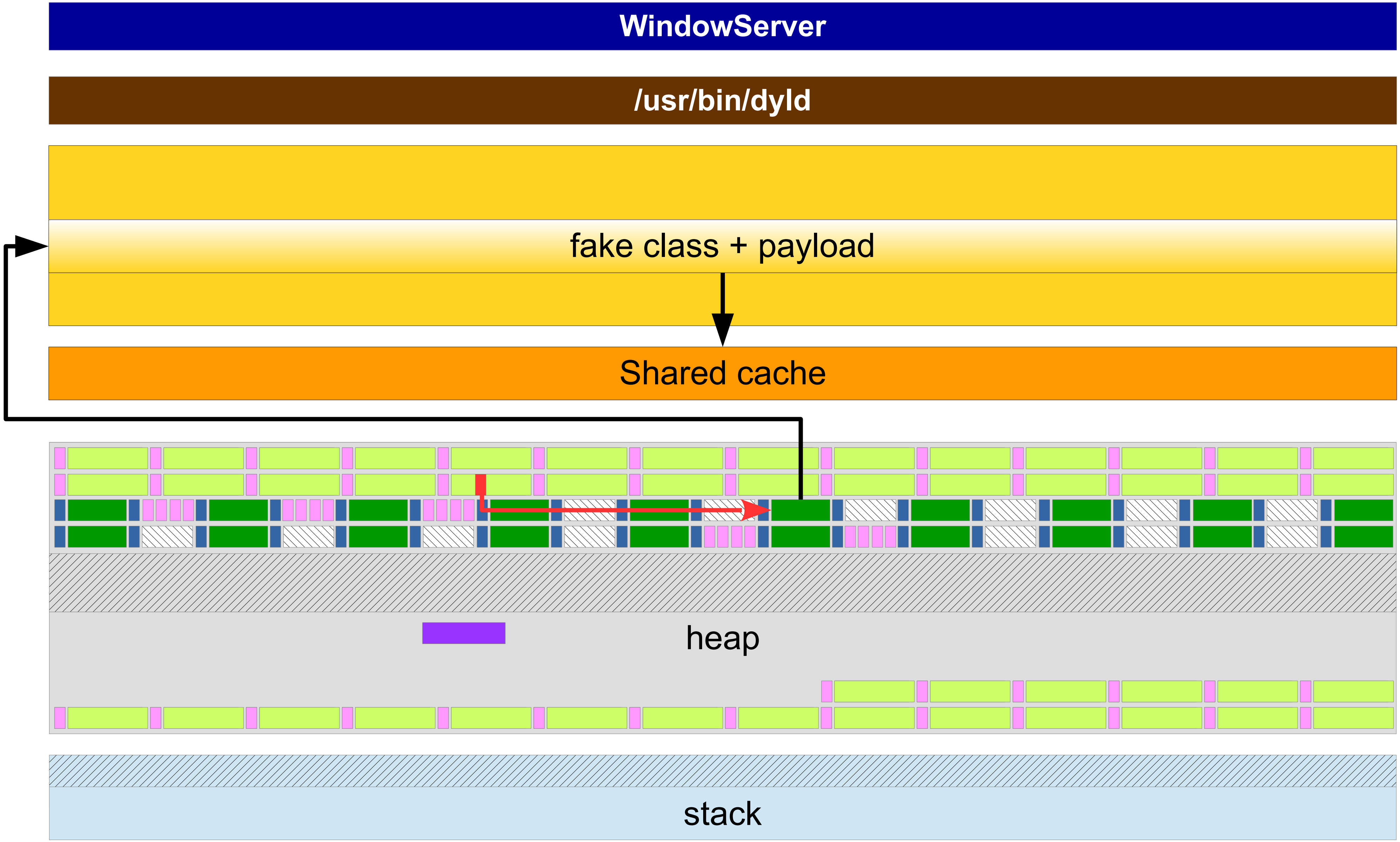
Memory reuse



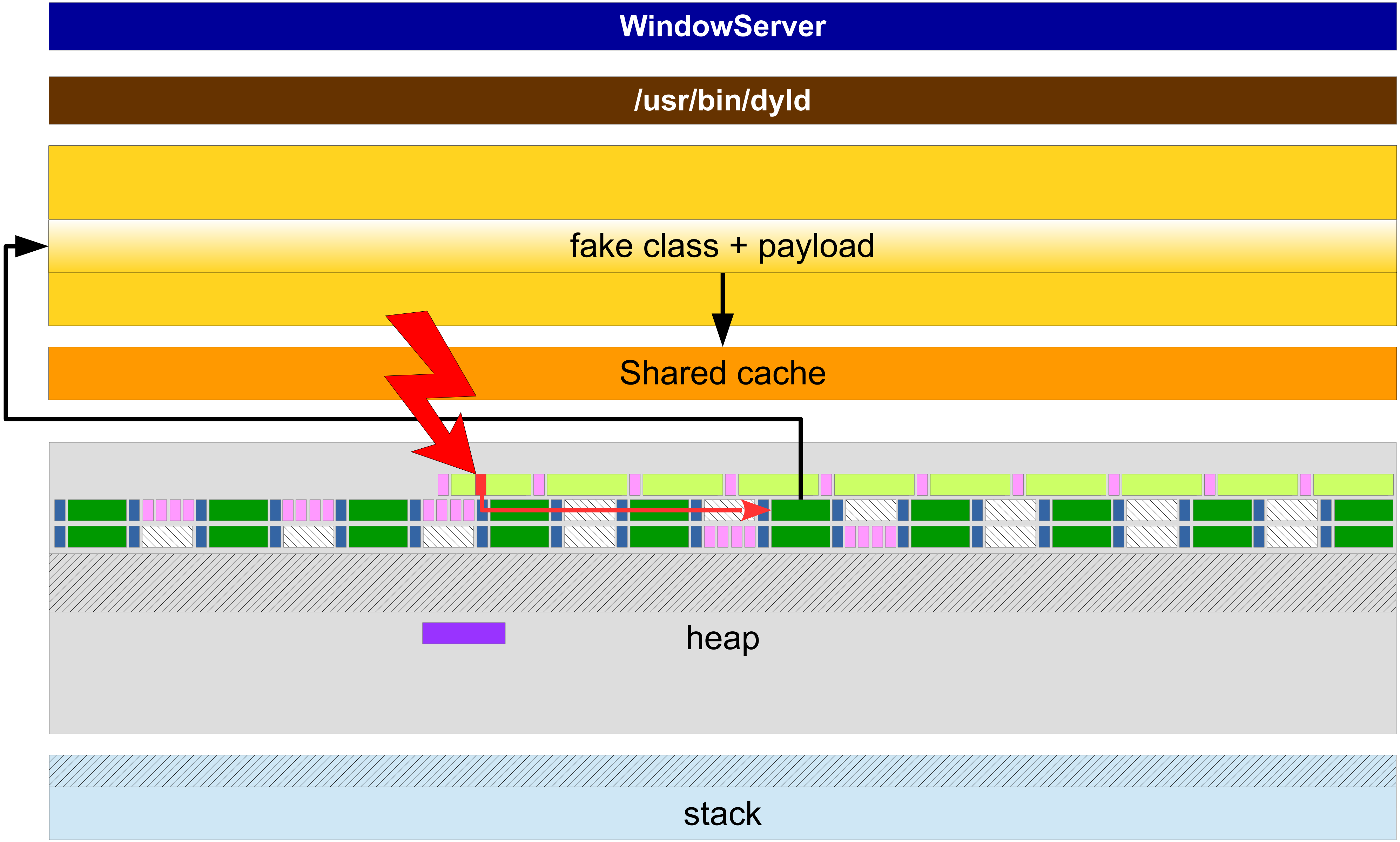
Redefinition of the property 1/4



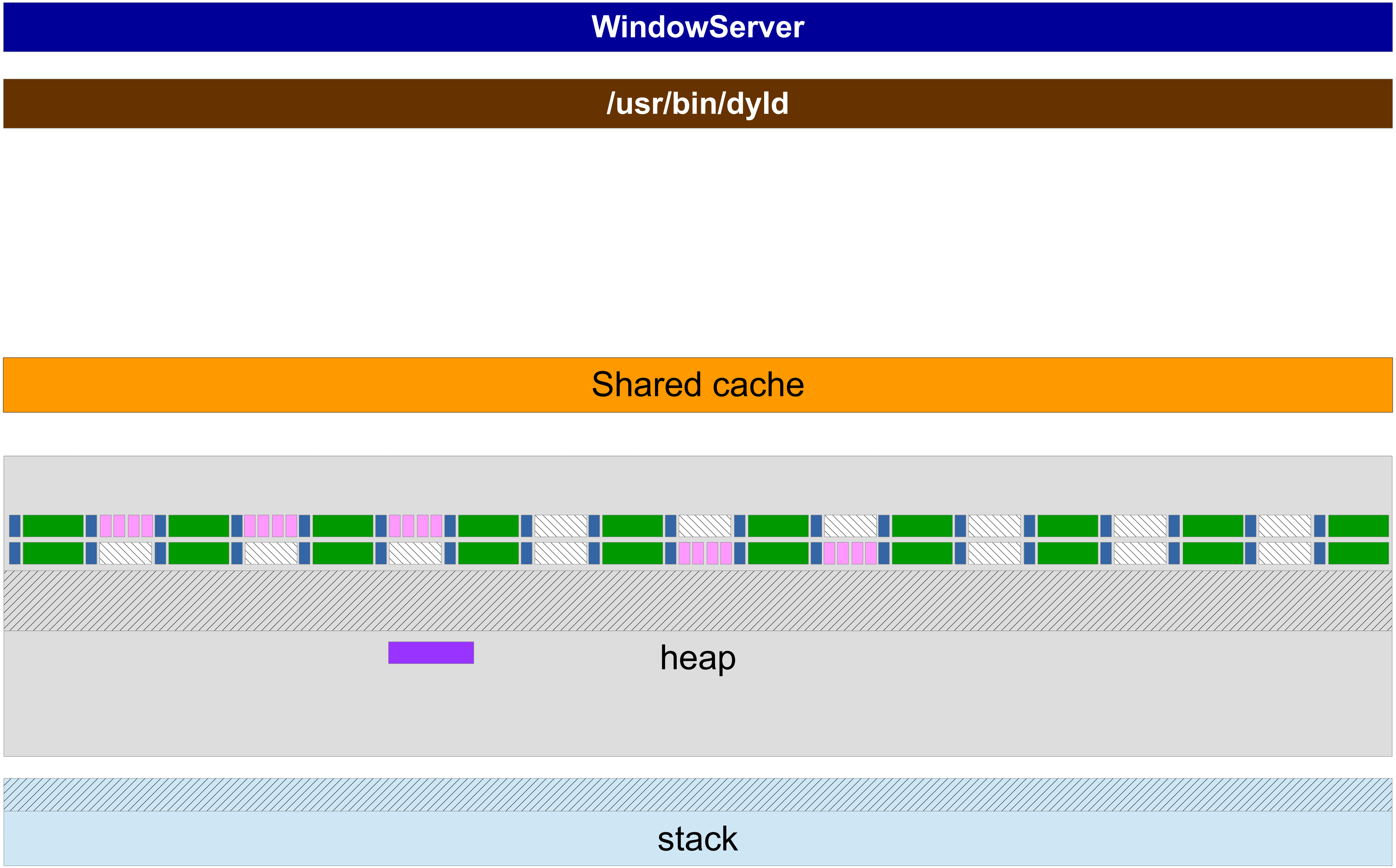
Redefinition of the property 2/4



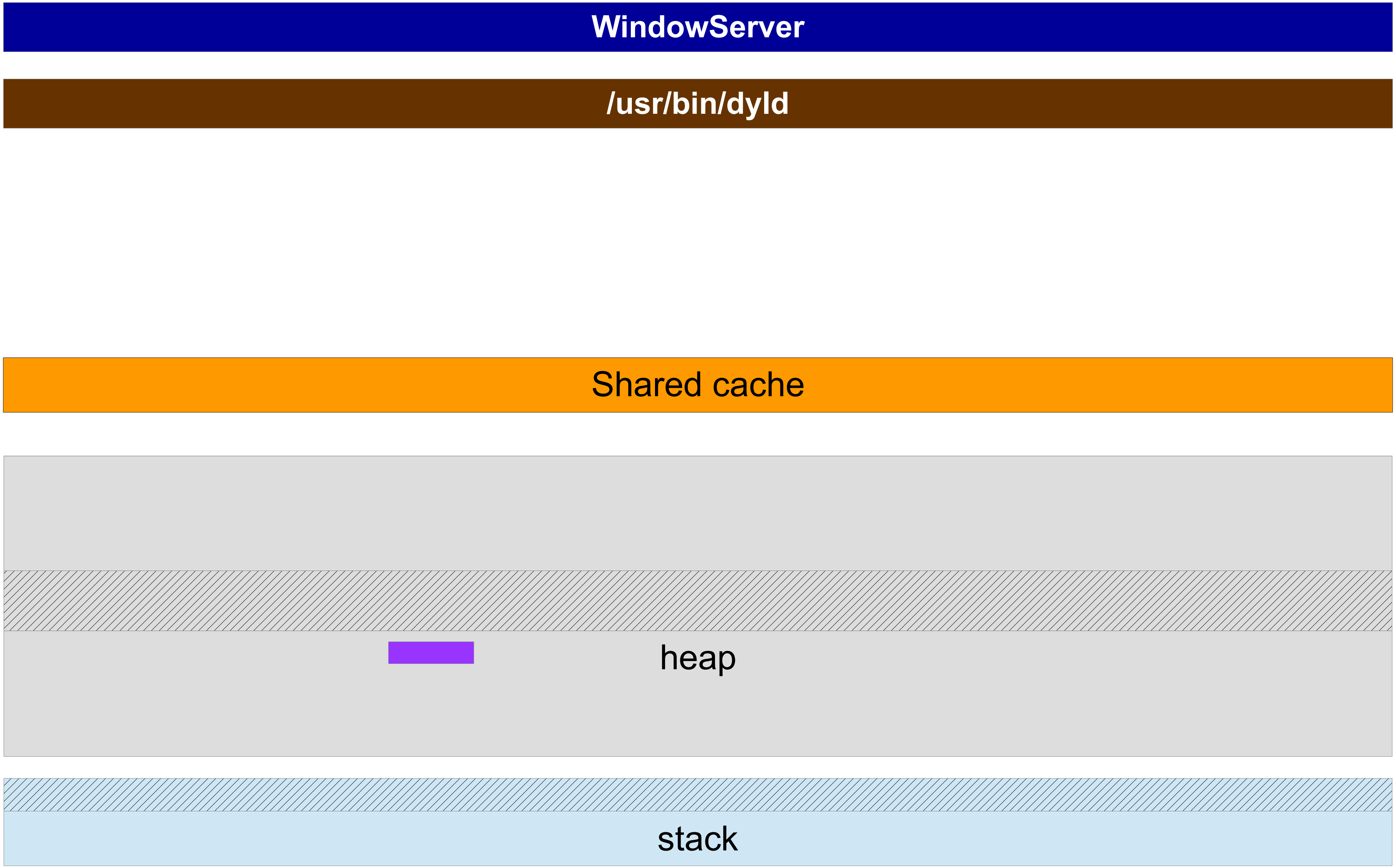
Redefinition of the property 3/4



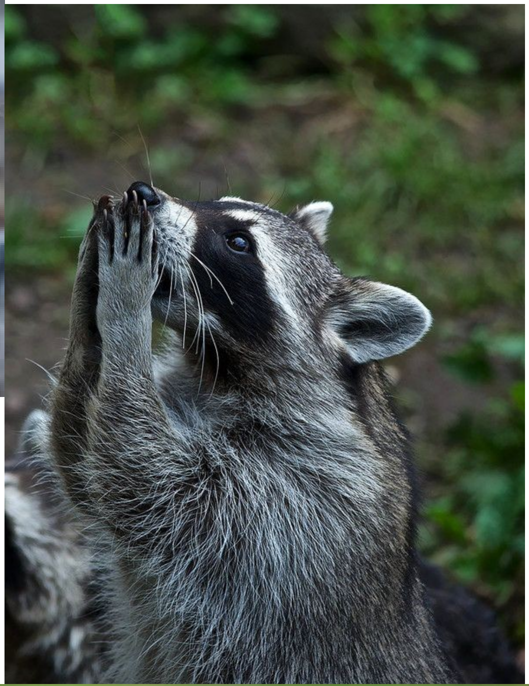
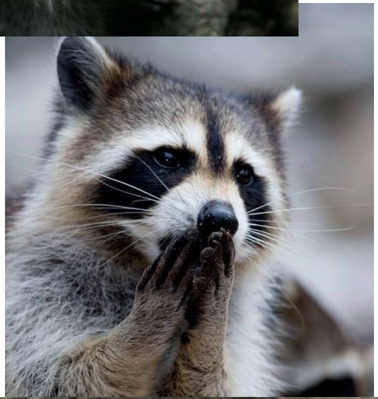
Redefinition of the property 4/4



Clean





Demo



Conclusion

- **Challenge**
  - Exploit takes ~8sec to execute arbitrary commands
    - And we could gain some more seconds by pre-serializing things
  - Exploit is very stable
    - The only thing that can fail is the allocation reuse
  - Only CVE-2018-4193 was used
- **Making good exploits takes time**
  - 18 days for just 900 line of code
  - But this is a lot of fun :)
- **Exploit source is available**
  - https://github.com/Synacktiv/CVE-2018-4193

Thanks

- **ret2 Systems**
  - For the opportunity to work on this
  - And the Binary Ninja commercial license!
- **Synacktiv**
  - For the time and the reviews
- **OffensiveCon**
  - For the amazing event :)
- **My wife**
  - For letting me go to Berlin on Valentine's Day
- **You**
  - For your attention!

Do you have any questions?

THANK YOU FOR YOUR ATTENTION

SYNACKTIV
DIGITAL SECURITY