# HEAP EXPLOIT DEVELOPMENT – CASE STUDY FROM AN IN-THE-WILD IOS 0-DAY

*September 4, 2019*

Last week, Google published a series of blog posts detailing five iOS exploit chains being used in the wild that were found by Google's Threat Analysis Group (TAG) team back in February. The reverse-engineering of how these exploits work by Google's Project Zero team is very complicated and detailed, and for professional exploit developers or platform security engineers, there's lots of information in these posts on how these specific exploits worked, along with how Google approached analyzing them.

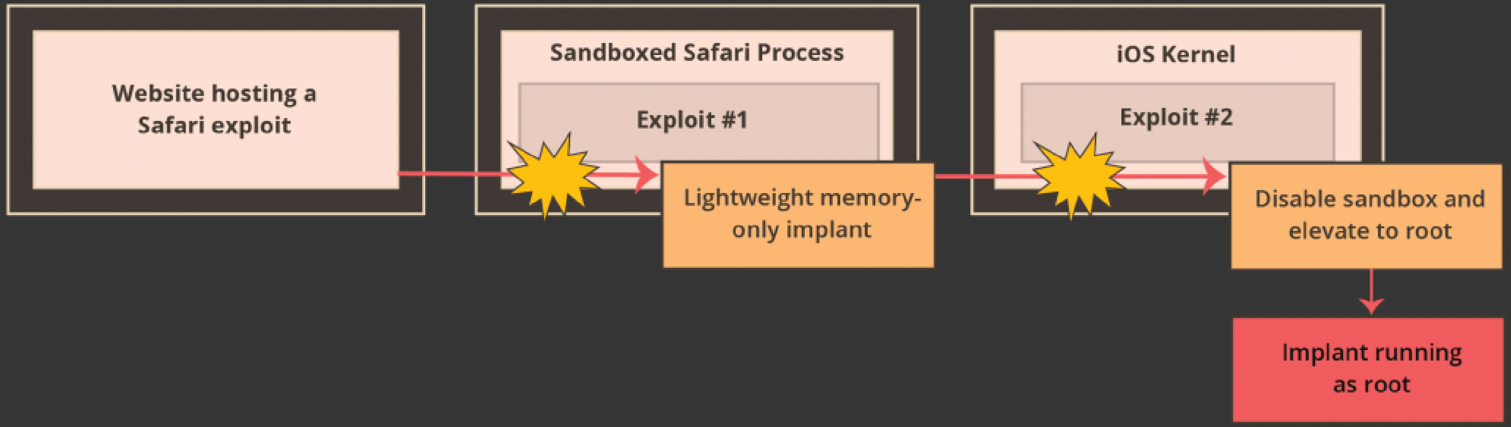
Between the technical details, hex dumps, and code-snippets lies a great story about the strategies and thought processes of the exploit developer themselves. Highlighting this story, and showing not just *what* the exploit developer did, but the strategy behind it might be helpful to people getting started at exploit development, as well as those who need to defend against memory corruption vulnerabilities in adjoining fields like platform security.

For this reason, I want to do a write-up of one of the iOS exploits that Google found with a slightly different focus. As we walk through the various stages of *what* the exploit is doing, we can also look at some major vulnerability categories and exploitation strategies in general, and hopefully by doing so we can pull back the curtain a little on the magic of modern memory-corruption exploits, and see the huge amount of engineering that enables exploit developers to turn software defects into reliable exploits.

In this first part of the series, I will give an overview of the different stages in this exploit chain. In the next few parts, we will cover the various stages of the exploit, and how it builds up tools (called "exploit primitives") that it can use to take full control of the operating system.

## EXPLOITS AND EXPLOIT CHAINS

Few modern exploits against hardened devices can fully compromise a device in a single go. Instead, exploit developers usually must *chain* multiple exploits together, each giving a remote attacker a little more leverage over the device, until the device is fully compromised. Each exploit is built around one or more software vulnerabilities, with large amounts of exploit engineering work so the vulnerability causes the target program to fail in a way the exploit developer can reliably control, rather than simply causing the target program to crash.


Website hosting a Safari exploit
Sandboxed Safari Process
Exploit #1
Lightweight memory-only implant
iOS Kernel
Exploit #2
Disable sandbox and elevate to root
Implant running as root

In our case, this exploit is the second part of a chain, and is intended to be paired with some other exploit targeting the phone's web-browser. That first exploit breaks into the web-browser from a webpage hosting that exploit. The exploit takes control of *Safari* and provides a platform from which *this* exploit can be launched against the iOS kernel itself. This exploit's central job is to disable the *Safari* sandbox, allowing full control of the device. This access is then used to disable various protections on the iPhone, such as the code-signing requirement and platform policy, before downloading and launching a second-stage implant in the background.

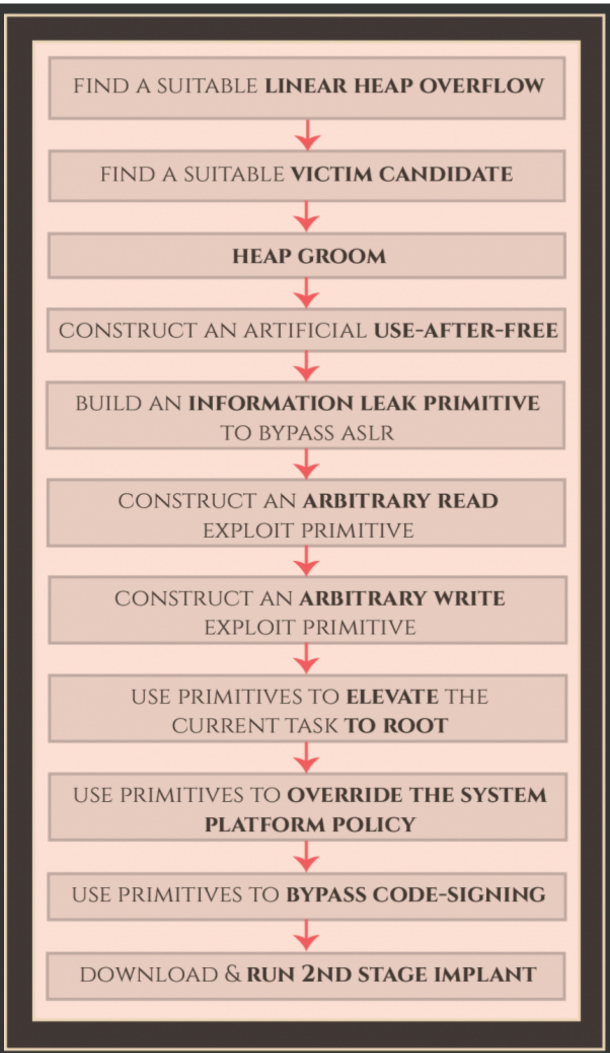
In general, this second-stage implant can do anything to the device. But in the specific case of this exploit chain being used in the wild, the implant is used to give interactive control over the phone, as well as steal chat history from WhatsApp, Telegram, iMessage, Google hangouts, the contents of emails from attached email accounts, contact lists, app lists, photos, device information including phone number, name, and IMEI, provide real-time GPS tracking of the target, and upload saved credentials, including Wi-Fi access point credentials from the phone. All of these are uploaded to the attackers' server, and the implant persists until the device is rebooted. A detailed breakdown of how the implant used in the attack works can be found here.

## THE IOS EXPLOIT, AND THE GENERAL EXPLOIT STRATEGY

The iOS kernel exploit used in these attacks is built around a *linear heap overflow* vulnerability in a graphics driver. The vulnerability itself is not particularly exciting, but it is exploited in a very interesting way: rather than exploiting the heap overflow directly, the exploit developer uses the vulnerability to construct a completely artificial *use-after-free* vulnerability which is easier to control and build into powerful exploit primitives. The exploit developer then uses these exploit primitives to surgically gain access to the kernel's private memory, which can be used to disable the devices' defences against malware. Having done so, the exploit downloads and installs a malicious implant as *root* on the device.

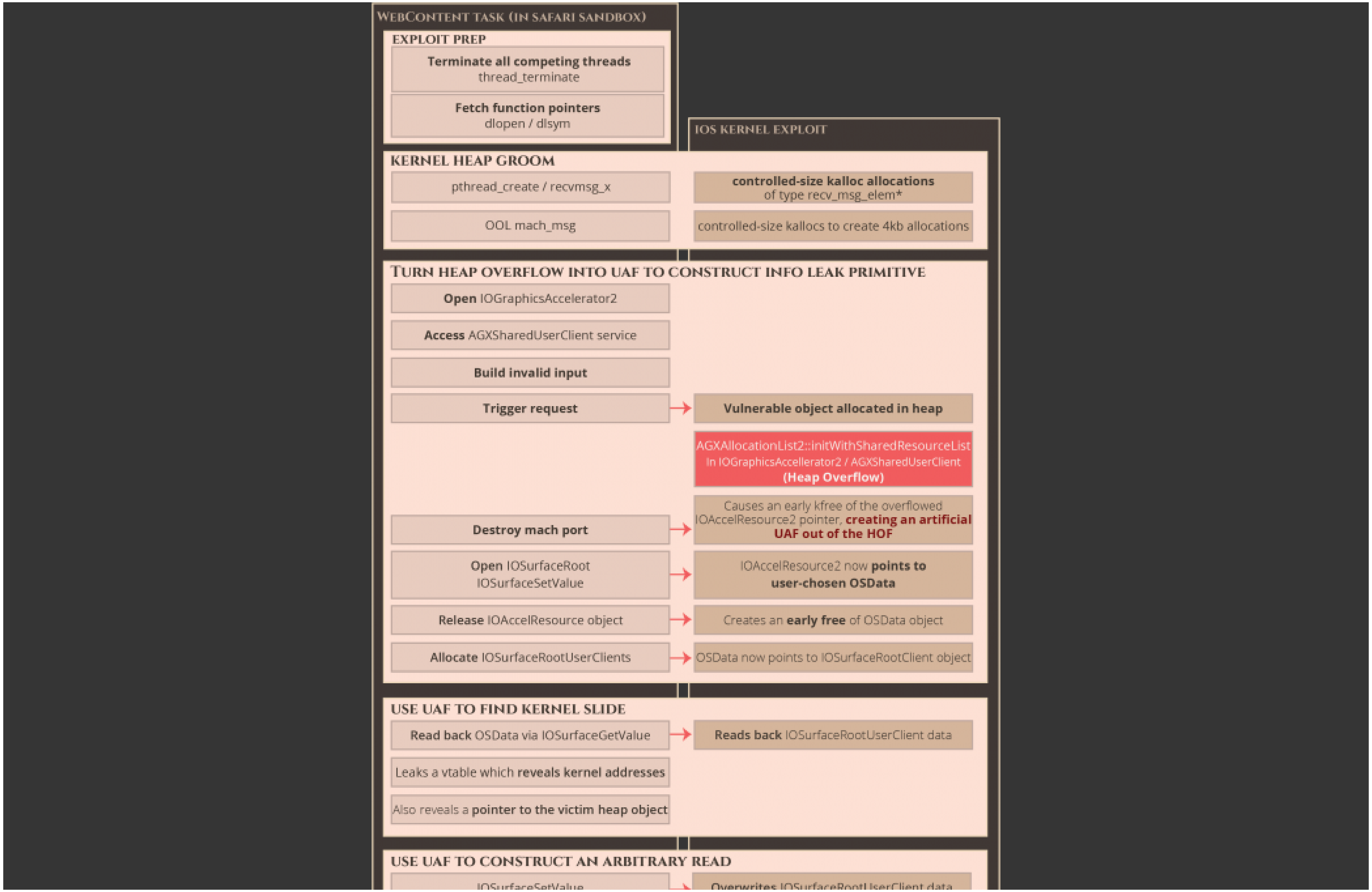
The hacker's overall strategy for exploiting this vulnerability is laid out below. Over the next few posts, I'll discuss each step in more detail, and we'll be able to see how exploit developers turn complex memory-corruption vulnerabilities into reliable exploits, even on modern hardened devices.

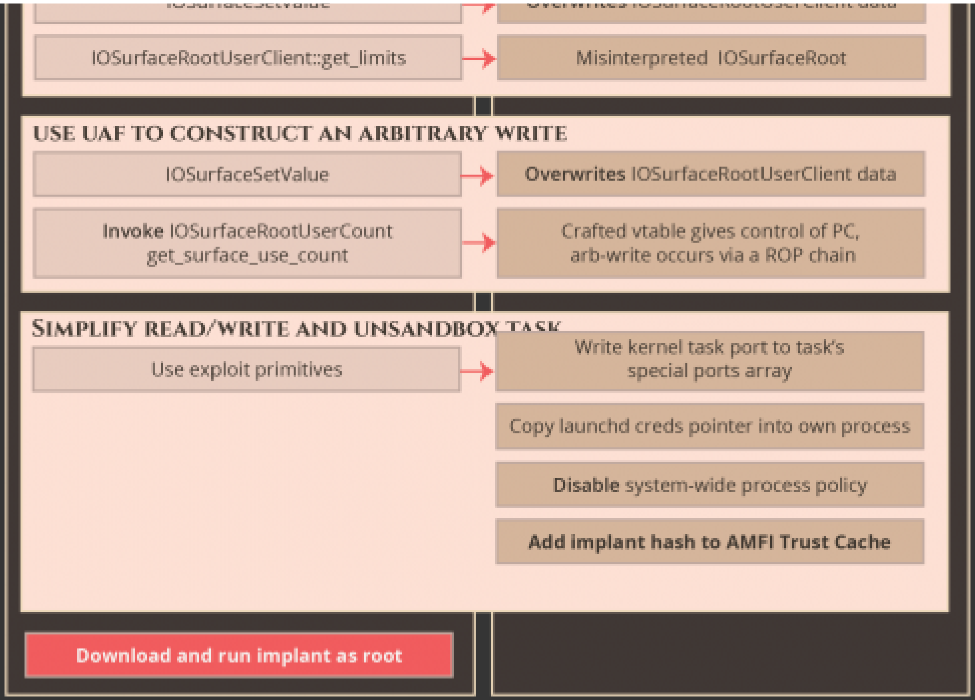
1. Find a suitable ***linear heap overflow*** around which the exploit can be built, and which is triggered on some *vulnerable object*
2. Find a suitable ***victim candidate*** that will be corrupted
3. Use a ***heap groom*** so the *vulnerable object* sits adjacent to a *victim object*.
4. Use a conversion technique to ***construct an artificial use-after-free*** that is easier for the exploit developer to control.


FIND A SUITABLE LINEAR HEAP OVERFLOW
FIND A SUITABLE VICTIM CANDIDATE
HEAP GROOM
CONSTRUCT AN ARTIFICIAL USE-AFTER-FREE
BUILD AN INFORMATION LEAK PRIMITIVE TO BYPASS ASLR
CONSTRUCT AN ARBITRARY READ EXPLOIT PRIMITIVE
CONSTRUCT AN ARBITRARY WRITE EXPLOIT PRIMITIVE
USE PRIMITIVES TO ELEVATE THE CURRENT TASK TO ROOT
USE PRIMITIVES TO OVERRIDE THE SYSTEM PLATFORM POLICY
USE PRIMITIVES TO BYPASS CODE-SIGNING
DOWNLOAD & RUN 2ND STAGE IMPLANT

5. Use the UAF to build an ***information leak primitive***, and leak kernel addresses, bypassing ASLR.
6. Use the UAF to construct an ***arbitrary read*** exploit primitive.
7. Use the UAF to build an artificial ***arbitrary write*** exploit primitive out of a *vtable overwrite* + ROP sequence.
8. Use the constructed primitives to elevate the current task to root, and disable system protections including the platform policy and code-signing requirements
9. Download and run the second-stage implant, which runs as root in the background.

A detailed view of the entire exploit is shown below. We'll cover each part in other posts, after which you will be able to click on the corresponding section for a detailed overview.


WebContent task (in Safari sandbox)
Exploit prep
Terminate all competing threads
thread_terminate
Fetch function pointers
dlopen / dlsym
iOS kernel exploit
Kernel heap groom
pthread_create / recvmsg_x
controlled-size kalloc allocations
of type recv_msg_elem*
OOL mach_msg
controlled-size kallocs to create 4kb allocations
Turn heap overflow into UAF to construct info leak primitive
Open IOGraphicsAccelerator2
Access AGXSharedUserClient service
Build invalid input
Trigger request
Vulnerable object allocated in heap
AGXAllocationList2::initWithSharedResourceList
In IOGraphicsAccellerator2 / AGXSharedUserClient
(Heap Overflow)
Causes an early kfree of the overflowed IOAccelResource2 pointer, creating an artificial UAF out of the HOF
Destroy mach port
Open IOSurfaceRoot
IOSurfaceSetValue
IOAccelResource2 now points to user-chosen OSData
Release IOAccelResource object
Creates an early free of OSData object
Allocate IOSurfaceRootUserClients
OSData now points to IOSurfaceRootClient object
Use UAF to find kernel slide
Read back OSData via IOSurfaceGetValue
Reads back IOSurfaceRootUserClient data
Leaks a vtable which reveals kernel addresses
Also reveals a pointer to the victim heap object
Use UAF to construct an arbitrary read
IOSurfaceSetValue
Overwrites IOSurfaceRootUserClient data


IOSurfaceRootUserClient::get_limits
Misinterpreted IOSurfaceRoot
USE UAF TO CONSTRUCT AN ARBITRARY WRITE
IOSurfaceSetValue
Overwrites IOSurfaceRootUserClient data
Invoke IOSurfaceRootUserCount get_surface_use_count
Crafted vtable gives control of PC, arb-write occurs via a ROP chain
SIMPLIFY READ/WRITE AND UNSANDBOX TASK
Use exploit primitives
Write kernel task port to task's special ports array
Copy launchd creds pointer into own process
Disable system-wide process policy
Add implant hash to AMFI Trust Cache
Download and run implant as root

## ARM Exploit Development

Writing ARM Shellcode

TCP Bind Shell (ARM 32-bit)

TCP Reverse Shell (ARM 32-bit)

Process Memory and Memory Corruption

Stack Overflow Challenges

Process Continuation Shellcode

Introduction to Glibc Heap (malloc)

Introduction to Glibc Heap (free, bins)

**Part 1: Heap Exploit Development**

Part 2 Heap Overflows and iOS Kernel

---

Twitter: @Fox0x01 and @azeria_labs

ARM Assembly Cheat Sheet

POSTER DIGITAL