rootlessJB – 11.0-11.3.1

Backstory

Back in the end of May 2018 Ian Beer (@i41nbeer) announced something that got everyone excited:

"If you're interested in bootstrapping iOS kernel security research keep a research-only device on iOS 11.3.1 for more tfp0. Release probably next week…"

This well-known security researcher did it again, a kernel exploit for iOS 11.3.1 was created. Developers & jailbreakers got all hyped up as he opened the door for a possible jailbreak to be built. A few days later, on June 5th, Ian tweets again saying that one from his two exploits was released, named "multi_path" and later in June 13th he released his second one, called "empty_list", slightly less reliable. **Update:** The bugs these exploits used are detailed at the end of the document, on the "Bonus" pages.

Immediately after those exploits @electra_team announced a 11.2-11.3.1 jailbreak and started working on it, however contrary to last time (Electra for iOS 11.0-11.1.2) all the development was closed to a group of developers and the team itself, supposedly to "prevent malware".
Teasings & screenshots, however weren't missing; those caused drama and made some people angry. I was totally against a closed-source jailbreak, especially when the jailbreak community was suffering, so I started working on my own, open-source and drama-free. I'm fairly new to the jailbreak scene & iOS itself (hint: started jailbreaking after iOS 9.3.3 and started programming in 2016), so this would also serve to my knowledge & I'd have great fun afterall 😋

Let's keep in mind what we're about to do

Over the years the meaning of "jailbreak" has slightly changed, from carrier unlocks, to getting filesystem access etc. However the core of it still holds the same. The basic things any modern jailbreak needs to have:

- Unsigned code execution: Making the system shut up and stop requiring certificate signing for everything we want to execute
- Lifted privileges, usually code execution as the root (most privileged) user & less sandbox restrictions
- Read & Write access to the root of the filesystem

And technically, every jailbreak up to iOS 10.3.3 (excluding Meridian) achieved all of these by patching the kernel. For unsigned code execution, they'd patch AMFI (AppleMobileFileIntegrity) to allow fakesigned binaries (signed without a valid certificate) to run, they'd patch the sandbox operations to allow any process to be able to *view* and load tweaks, by patching the file access and memory mapping restrictions, then they'd have to do a mount patch to allow the root partition to be remounted as read and write. There are different ways to remount the root partition. To help understanding this we have to understand the checks which we'll discuss later.

What we can and can't do

First let's consider our worst enemy since iOS 9: KPP (Kernel Patch Protection).
KPP keeps checking the kernel for changes every few minutes, when device isn't busy.

That "check every now and then" thing doesn't sound too good for a security measure, and in fact a full bypass was released by Luca Todesco and it involves a design flaw.
KPP does not prevent kernel patching; it just keeps checking for it and if one is caught, panics the kernel. However, since we can still patch, that opens up an opportunity for race conditions. If we do things fast enough and then revert, KPP won't know anything 😏

In a nutshell with every kernel patch, smartly enough Luca Todesco creates a fake copy of the kernel pages, redirects execution there, makes patches and lets KPP check the original pages which, are unmodified! (For a more in-depth and detailed write-up check out this by Jonathan Levin)

When this bypass was released however Apple had already overcome this issue, they introduced KPP's big but younger brother: KTRR (Kernel Text Read-Only Region). Note the "Read-Only", Apple got rid of constant checking, they added protection on the hardware level instead which maps certain memory as read-only after being used. No ability to race this time, KTRR just won't let you write at the first place. A partial bypass was released by the same Luca Todesco but it was shortly patched in iOS 10.2.

Now, what to do? No bypass? No kernel patches. But what kind of patches to be exact? Hmm…

Yes, KPP & KTRR are supposed to keep the kernel unchanged, but can a program work if its memory is 100% static? No ability to write memory = no ability to store ANY kind of *variable* data. Thus the obvious conclusion is that these mechanisms protect only specific parts of the kernel, they protect the code that get executed and the constant data, which doesn't need to change, but they keep the variable data writeable.

Let's examine everything slowly.

1) sandbox policies: once upon a time we could do it but Apple decided to make them read-only. No reason to be writable at the first place.
2) setuid(0) (root): normally needs a sandbox patch
3) Unsigned code execution: AMFI policies never change either
4) r/w in /. As mentioned, we need to take a more in-depth look at this patch

Definition of kppless

So far, it might seem that we're out of luck. We can't do the basis. Or can we? There are jailbreaks that do it out there: Meridian? Electra? LiberiOS?

So, if we can't patch the rules of those security policies such as AMFI and sandbox *in order to allow lower privilege processes to do something*, guess it: why don't we patch the apps themselves *to give them more privileges?* Not every process is as restricted as user apps on iOS, so the kernel has to keep track of what privileges to give to them, not all restrictions are automatically applied on everything. Processes are launched and killed, they don't stay forever in memory and by taking a look at how the kernel knows what process X can and can't do, we conclude to the fact that their privileges can change. That's exactly the meaning and the core of "kppless" jailbreaks: not adapting the kernel to fit our needs, but adapting ourselves to make the kernel happy. This technique and name was originally proposed by xerub.

Nevertheless this causes many challenges which we'll discuss later on ;)

More understanding of kppless

The kernel stores most writable data in structures. In memory all elements of a structure are aligned one after another. The address of the structure is also the address of the first element of it. The second element is at the structure's address + sizeof(first element) and so on by adding the size of previous elements. Here's a simple example of reading & writing on a struct using pointers and offsets:

```
struct my_structure {
    uint64_t first_element; // 64 bits = 64/8 bytes = 8 bytes
    uint32_t second_element; // 32 bits = 32/8 bytes = 4 bytes
    char *third_element; // 64 bits = 64/8 bytes = 8 bytes
};


int main(int argc, const char * argv[]) {

    struct my_structure mystruct;

    unsigned int *addr = (unsigned int*)&mystruct; // get address of structure
    uint64_t *addr_first_element = (uint64_t *)addr; // address of first element = address of
struct
    uint32_t *addr_second_element = addr + sizeof(mystruct.first_element); // address of
first + size of it
    char *addr_third_element = (char *)addr_second_element + sizeof(mystruct.second_element);

    // get some data ready
    uint64_t first = 0x4141;
    uint32_t second = 0x4242;
    char third[] = "CCCC";

    // copy them to our retrivied addresses
    memcpy(addr_first_element, &first, sizeof(first));
    memcpy(addr_second_element, &second, sizeof(second));
    memcpy(addr_third_element, &third, sizeof(third));

    // read the data back
    printf( "element 1: 0x%llx\n"
            "element 2: 0x%x\n"
            "element 3: %s\n",
            *addr_first_element,
            *addr_second_element,
             addr_third_element
          );
    return 0;
}
```

Offsets

On the kernel we have to calculate the struct offsets manually. This can be done in many ways. We can calculate them by hand, we can disassemble functions referencing to those struct members, or we could also grab the headers from the XNU source code (need to thank Apple for open sourcing that) and try our luck in including them into a compilable project, from there just use the built-in function 'offsetof(struct type, member)'.

Disassembling

- Find a reference of the struct member you're looking for in an open-source function from XNU
- Get the kernelcache off your device using Apple File Conduit "2" or Cyberduck then decompress it using lzssdec (Update: on iOS 12 you gotta get it off the ipsw/OTA package; no more read access to kernelcache)
- Put it a disassembler such as Hopper
- Find the same open-source function (if you can't find it, well not everything is symbolicated, lookup another one)
- Find the same reference of that, which in this case will not be like "struct.member" or "struct->member" but like I showed on the previous example, using pointers, so like this "*(struct + offset)"

Finding offsets manually:

- First, navigate to the XNU source code by Apple
- Find the desired header
- Calculate the size of everything *before* our target
- If there's a type you don't really recognize, you need to find where's that type defined, usually just google "typedef unknown_type" and it works :P
- That's your offset

Including headers

- Create a header and copy the struct definition from the header on the XNU source code,
- Also add all type definitions that you can probably find in other headers (-_-)
- Include header in project and do offsetof(struct, member)

Let's start patching… ourselves

So, the most basic thing you can start with would be getting root. All of our process info gets stored in the so-called "proc structs" (as seen in this XNU header). Getting root would be considered a simple task, we just need to override our user id in kernel. Don't get lost, p_uid member is not important, the user id that actually gets used, is stored in another structure, the ones storing our credentials, called "struct ucred" (header).
The ucred structure itself is a member of the proc structure (a pointer to "struct ucred" is also named as "kauth_cred_t").

So in short terms:

- Find our proc structure in kernel
- Calculate the ucred structure from there
- Calculate cr_uid (& all that stores a copy of our user id) in it (and optionally cr_gid; if you want 'wheel' group id)
- Write 0 in all of them

In order to find the proc structure in kernel we can either find "allproc" using patchfinding or kernproc using symbol finding (_kernproc symbol). From there start iterating and checking the pid.

If we chose the first way:

- Find allproc
- Read 8 bytes from it
- The result is the proc struct of the *latest* spawned process
- From there start iterating over all the structs. The first member (which is at the same address of the struct itself) of the proc struct will point to the previous spawned process
- If pid == getpid() then we found our proc struct. Pid can be found at offset 0x10 (16 bytes) from the proc struct

If we choose the second way:

- Find _kernproc symbol (Update: *Some* devices got 0 symbols on iOS 12 so that might not be an option on the future)
- Read 8 bytes from it
- The result is the kernel proc struct, which is the very first process
- From there start iterating over all the structs. The second member (since a 64bit pointer is 8 bytes, the second member is at the address of the struct + size of first member; so address + 8 bytes) of the proc struct will point to the *next* process
- If pid == getpid() then we found our proc struct

Code examples. First method:

```
uint64_t proc_of_pid(pid_t pid) {
    uint64_t proc = KernelRead_64bits(Find_allproc()); // Start from latest proccess
    pid_t pd; // For storing result pid

    while (proc != 0) { // iterate over all processes
        pd = KernelRead_32bits(proc + off_p_pid); // offset of p_pid = 0x10
        if (pd == pid) return proc; // Found it!
        proc = KernelRead_64bits(proc + off_p_prev); // Previous process. Offset = 0
    }
    // not found
    return 0;
}
```

Second method:

```
uint64_t proc_of_pid(pid_t pid) {
    uint64_t proc = KernelRead_64bits(find_symbol("_kernproc")); // Start from first proccess
    pid_t pd; // For storing result pid

    while (proc != 0) { // iterate over all processes
        pd = KernelRead_32bits(proc + off_p_pid); // offset of p_pid = 0x10
        if (pd == pid) return proc; // Found it!
        proc = KernelRead_64bits(proc + off_p_next); // Go to the next process. Offset = 8
    }
    // not found
    return 0;
}
```

Once we've found our proc struct we need to get into the ucred struct, which can be found 256 (0x100) bytes after the proc struct. We simply read 8 bytes of (ourProcStruct + 0x100) and that's where our credentials are stored. So:

```
void rootify(pid_t pid) {
    uint64_t proc = proc_of_pid(pid);
    uint64_t ucred = KernelRead_64bits(proc + off_p_ucred); // 0x100

    KernelWrite_32bits(ucred + off_ucred_cr_uid, 0); // 0x18
    KernelWrite_32bits(ucred + off_ucred_cr_ruid, 0); // 0x18 + 4 = 0x1c
    KernelWrite_32bits(ucred + off_ucred_cr_svuid, 0); // 0x1c + 4 = 0x20

    KernelWrite_32bits(ucred + off_ucred_cr_rgid, 0); // 0x68
    KernelWrite_32bits(ucred + off_ucred_cr_svgid, 0); // 0x68 + 4 = 0x6c
}
```

Next step: sandbox?

Ok so we've got root. Is that enough? Probably not ¯_(ツ)_/¯ : "file system sandbox blocked…", haha I know right! Sandboxing has taken over iOS, it's one of the biggest security measures Apple has implemented, nobody cares about uid: 0 if we're sandboxed. So let's dig in.

Now, probably almost everyone has chosen a lazy way to do this, replacing our credentials with the kernel's ones. Sure, that'll make you omnipotent (almost…) but is it really the best way to do this? Perhaps not, and if you do that you'll need to clean up and reverse it before the process exits, or say hello to a kernel panic… Let's take a more in-depth look at the ucred struct.

Near the end of it we can notice this:

```
struct label *cr_label; /* MAC label */
```

If "MAC" doesn't sound familiar to you (no, not Mac computers) then here I'm saying it: "Mandatory Access Control". MAC policies are what force codesigning rules, and… sandboxing, exactly what we need!

Let's find the definition of this label struct. It's defined on this file. There we can see a "l_perpolicy" array (offset 0x8) which defines various policies. Hmm let's override that with kernel's one instead! Nah! Won't mess with the kernel. Let's dig in.

The first element of it is the AMFI slot (at 0x8), and the second (at 0x10 or 16 in decimal) is the sandbox one. A quick comparsion between a sandboxed and not sandboxed process we can see that the sandboxed process contains a pointer in the sandbox slot, the unsandboxed one contains nothing. So let's do what a sensible man would do, let's nullify that pointer!

Not surprisingly, it works. You don't get as much power as you would with kernel creds, but you're out of sandbox and that's enough

```
void unsandbox(pid_t pid) {
    uint64_t proc = proc_of_pid(pid); // pid's proc structure on the kernel
    uint64_t ucred = KernelRead_64bits(proc + off_p_ucred); // pid credentials
    uint64_t cr_label = KernelRead_64bits(ucred + off_ucred_cr_label); // MAC label at 0x78

    KernelWrite_64bits(cr_label + off_sandbox_slot, 0); // nullify it, offset 16

}
```

Unsigned code execution anyone?

Basically every jailbreak with a KPP bypass patched AMFI in the kernel to simply “turn off” the protection. That would be done by setting the “cs_enforcement_disable” boot-argument which *can* be patched, but on top of that PE_I_can_has_debugger (debugging mode) needs to be enabled otherwise the kernel will panic (see image). That is unfortunately in a read-only area. What do we do?

```
if ( (unsigned int)PE_parse_boot_argn("cs_enforcement_disable", &v9, 4LL) )
{
  if ( v9 )
  {
    IOLog("%s: cs_enforcement disabled by boot-arg\n");
    if ( !(unsigned int)PE_i_can_has_debugger(0LL) )
      panic(
        "\"can't has cs_enforcement_disable\"@/Library/Caches/com.apple.xbs/Sources/AppleMobileFileIntegrity/AppleMobil"
        "eFileIntegrity-270.50.2/AppleMobileFileIntegrity.cpp:2215");
  }
}
```

Well, Apple was kind enough to perform the codesigning checks in userland, by a daemon called “amfid” using a library called “libmis”. Since we’re in userland there are no KPP/KTRR-like limitations, (~~coming in A12 supposedly?~~ There is in A12. R.I.P… well this method which is not the only one) so we’re free to patch it and perform validation ourselves.

The point of AMFI patching has never been to get “unsigned” binaries to run, rather “fake-signed” binaries, so we don’t have to disable any check (which would be impossible in userland only), we just have to alter the existing checks so all kinds of signatures are supported.

There are two public ways to do this:

One, originally implemented by Ian Beer: the exception ports technique, consisting of:

1. Overwriting the method that performs checks with some dummy data, which would make amfid crash
2. Setting an exception handler function in our own process; so when amfid crashes it’s frozen and we have control over its memory
3. Perform the checks and calculate the signature blob hash ourselves
4. Resume execution of amfid

Two, originally used by ninjaprawn and later in Electra:

2. Creating a dynamic library which replaces the validation function with a custom one and inject that to amfid. That’s it. You know what to do later. Well, how do you inject the library to patch codesign if it’s not signed?
1. Add that library on the dynamic trustcache…

(hey, what’s a trustcache?)…

Unsigned code execution anyone?

For speed purposes Apple caches the signature hashes of every Apple-binary of iOS in the so-called “trustcaches”. If kernel sees that the code-directory hash of the binary is on the trustcache it doesn’t perform further validation, it just assumes the binary is “trusted” and allows code execution. The rest are handled by amfid. Obviously that is static, binaries aren’t supposed to change, what’s so “dynamic” about it? Well Apple isn’t worried just about built-in binaries, if you’ve ever used Xcode you probably noticed a “Developer” menu pop up in the Settings up. Xcode sends to your device a dmg image full of binaries to help app debugging. Apple didn’t do what you would think they did, if the dmg image itself is properly signed the device will trust ALL the binaries in another trustcache, yes, that is dynamic! We can patch it and trust our own stuff!

Xerub was the first to reveal this technique publicly as seen in his kppless fork of extra_recipe.

It’s a little bit similar to the amfid patch technique:

- Find the dynamic trustcache in kernel
- Create a fake “trustchain” struct
- Calculate the codesign hashes of the binary

```
    uint64_t trust_chain = Find_trustcache(); // find the trustcache

    struct trust_chain fake_chain; // fake struct
    fake_chain.next = KernelRead_64bits(trust_chain); // pointer to original chain
    *(uint64_t *)&fake_chain.uuid[0] = 0xabadbabeabadbabe; // always like this
    *(uint64_t *)&fake_chain.uuid[8] = 0xabadbabeabadbabe; // always like this

    int cnt = 0;
    uint8_t hash[CC_SHA256_DIGEST_LENGTH]; // store hash
    hash_t *allhash = malloc(sizeof(hash_t) * [paths count]); // 20 bytes each hash
    for (int i = 0; i != [paths count]; ++i) {
        uint8_t *cd = getCodeDirectory((char*)[[paths objectAtIndex:i]
UTF8String]); // find code directory
        if (cd != NULL) {
            getSHA256inplace(cd, hash); // sha256 the code directory
            memmove(allhash[cnt], hash, sizeof(hash_t));
            ++cnt;
        }
        else continue;
    }

    fake_chain.count = cnt;

    size_t length = (sizeof(fake_chain) + cnt * sizeof(hash_t) + 0xFFFF) & ~0xFFFF;
    uint64_t kernel_trust = Kernel_alloc(length); // allocate data in kernel
    KernelWrite(kernel_trust, &fake_chain, sizeof(fake_chain)); // override
    KernelWrite(kernel_trust + sizeof(fake_chain), allhash, cnt * sizeof(hash_t));
    KernelWrite_64bits(trust_chain, kernel_trust);
    free(allhash);
```

Now you choose either method. I'm not going to discuss this further but you can check out the source codes, first and second.

Now, there are some other complications. We need amfid's task port in order to be able to patch it. This is achieved either by task_for_pid-allow entitlement on ourselves or get-task-allow on amfid.

To do that we either steal creds from another process (lazy way) or patch our own (or amfid's), let's see how.

Normally entitlements are cached in our vnode struct (a vnode is a structure the kernel generates for every file). We get our vnode from proc->p_textvp then we go to ->vu_ubcinfo->cs_blobs->csb_entitlements_blob and patch that. However since the process is already launched, some entitlements might also be cached elsewhere, and that's exactly what happens with task_for_pid-allow and get-task-allow. They're cached on the AMFI slot of cr_label, differently from the csblob ones they're unserialized, thus stored in kernel objects as an OSDictionary and not in raw XML. To unserialize we use kernel calls.

```
void entitlePidOnAMFI(pid_t pid, const char *ent, BOOL val) {
    uint64_t proc = proc_of_pid(pid);
    uint64_t ucred = KernelRead_64bits(proc + off_p_ucred);
    uint64_t cr_label = KernelRead_64bits(ucred + off_ucred_cr_label);
    uint64_t entitlements = KernelRead_64bits(cr_label + off_amfi_slot); // 0x8

    if (OSDictionary_GetItem(entitlements, ent) == 0) { // check if it's there
        OSDictionary_SetItem(entitlements, ent, (val) ? Find_OSBoolean_True() :
Find_OSBoolean_False()); // set as true or false
    }
}
```

Now simply give amfid "get-task-allow" as true or give ourselves "task_for_pid-allow" and "com.apple.system-task-ports", both as true.

For more advanced entitlements we can construct an XML string, unserialize using OSUnserializeXML function on kernel and write the address into the AMFI slot.

Nevertheless, some entitlements are read from the cached blob in our vnode. To patch that we need to:

- Find our proc->p_textvp->vu_ubcinfo->cs_blobs->csb_entitlements_blob
- Put our custom XML in the first blob->data
- Update the signature hash
- Add unserialized entitlements in proc->p_textvp->vu_ubcinfo->cs_blobs->csb_entitlements

CSBlob-ent patching?

Time for some code, here you go: (all error handling code & printf()s are removed to keep the code in one page)

```
BOOL patchEntitlements(pid_t pid, const char *entitlementString) {

    if (!pid) return NO;

#define SWAP32(val) __builtin_bswap32(val)

    struct cs_blob *csblob = malloc(sizeof(struct cs_blob));
    CS_CodeDirectory *code_dir = malloc(sizeof(CS_CodeDirectory));
    CS_GenericBlob *blob;

    uint64_t proc = proc_of_pid(pid);
    uint64_t vnode = KernelRead_64bits(proc + off_p_textvp);
    uint64_t ubc_info = KernelRead_64bits(vnode + off_v_ubcinfo);
    uint64_t cs_blobs = KernelRead_64bits(ubc_info + off_ubcinfo_csblobs);

    KernelRead(cs_blobs, csblob, sizeof(struct cs_blob)); // read from there into the csblob struct

    uint64_t codeDirAddr = (uint64_t) csblob->csb_cd;
    uint64_t entBlobAddr = (uint64_t) csblob->csb_entitlements_blob;

    KernelRead(codeDirAddr, code_dir, sizeof(CS_CodeDirectory)); // read into the code directory struct

    // get length of our current blob; use SWAP32 to convert big endian to little endian
    uint32_t length = SWAP32(KernelRead_32bits(entBlobAddr + offsetof(CS_GenericBlob, length)));

    blob = malloc(sizeof(CS_GenericBlob)); // allocate space for our new blob

    KernelRead(entBlobAddr, blob, length); // read that much data into the CS_GenericBlob struct

    uint8_t entHash[CC_SHA256_DIGEST_LENGTH];
    uint8_t digest[CC_SHA256_DIGEST_LENGTH];

    // add our new entitlements
    sprintf(blob->data, "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<!DOCTYPE plist PUBLIC \"-//Apple//
DTD PLIST 1.0//EN\" \"http://www.apple.com/DTDs/PropertyList-1.0.dtd\">\n<plist
version=\"1.0\">\n<dict>\n%s\n</dict>\n</plist>\n", entitlementString);

    CC_SHA256(blob, length, digest); // calculate the SHA256

    KernelWrite(codeDirAddr + SWAP32(code_dir->hashOffset) - CSSLOT_ENTITLEMENTS * code_dir->hashSize,
digest, sizeof(digest)); // write our new hash

    free(code_dir);

    // write our new blob
    KernelWrite(entBlobAddr, blob, length);

    bzero(blob, sizeof(CS_GenericBlob));

    // Add unserialized entitlements too
    uint64_t newEntitlements = OSUnserializeXML(blob->data);

    KernelWrite_64bits((uint64_t)csblob->csb_entitlements, newEntitlements);

    free(csblob);
    free(blob);
    return (entitlements == newEntitlements) ? YES : NO;
}
```

All set! Drop some binaries, start SSH & code injection

Now, time to drop some binaries. For that we need to remount the root partition as read & write. Sandbox doesn't allow code execution in writable paths.

As said earlier, let's discuss how. mount() is a system call, code gets ran by the kernel. During its execution, mount() calls mount_common() (code in here), which calls mac_mount_check_remount (reference) which then triggers the sandbox hook mpo_mount_check_remount(), which checks if the MNT_ROOTFS flag is set on the vnode of / (root), and if so, disallows the mount.

So a way to overcome that would be patching the checks, but since that is not possible, we can instead remove the MNT_ROOTFS flag temporarily and mount. Remember, vnodes are mutable structures.

Luca Todesco (not xerub as you might have heard) was the first to come with the approach:

1. Get vnode of / (which is mount point of /dev/disk0s1s1; the root partition)
2. Patch mount flags that specify the fact that it's read-only and a root-filesystem
3. Make an "Update" mount() of /dev/disk0s1s1 on /
4. Patch flags back just in case
5. Done! We have r/w

A quick note however: This check is **not** the only one on iOS 10.2. Luca Todesco also patched the sandbox hooks, with his KPP bypass. After APFS (10.3) less checks are made and that method is enough to remount. Since / and /private/var share their container and the latter needs to be writable no more checks are being done by Lwvm (Lightweight-volume-manager). Stek29 discussed a kppless way of remounting / pre-APFS on this blog, which has been used in the Meridian jailbreak.

However try that method on iOS 11.3+ and after getting success from the mount() call realize whenever you try to write something in / the kernel panics. What's going on? Spark Zheng & Xiaolong Bai who had been working on a private jailbreak way before any 11.3 exploit was public, decided to release their write-up on how they got r/w in /

If you try to run "/sbin/mount" (a default Apple binary) from the output you get you'll realize the mounted device in / isn't /dev/disk0s1s1 but rather something with a long hash on its name *and* "/dev/disk0s1s1". It's an APFS snapshot, those were introcuded with iOS 10.3 and are basically read-only copies of a mount device. By default on 11.3 a snapshot of / is created and mounted in /, and restored after every reboot. Since a snapshot is not a valid mount point, if we try to write into it undefined behavior will trigger.

Remounting

The bypass of them was another lazy-method suitable for a developer jailbreak:

1. Mount /dev/disk0s1s1 somewhere in a writable path in /private/var
2. Copy mount data from the vnode of that path into the vnode of /
3. / will now act as that path and you can write in there.

Kind of hacky but only option :/ So I decided to give it a try. Not only I never got that to work (project was public and other people apparently did) but even coolstar himself when working on Electra said that the bypass was too buggy and made a device practically unusable. No surprises, swapping random pointers is not always a good idea if you want stability. Plus, since snapshot gets restored again after reboot, this was not a persistent bypass. Every file would get erased to stock. (And in fact, that is how Rollectra works if you're curious)

So at that point I could either give up, or… accept it. I had nothing to do so chose the second option. Who cares if we don't have write access in / ? :P

By time, Electra released and it did have a proper remount. Umang Raghuvanshi found that there was no check whatsoever about what snapshot was being restored, we could put an arbitrary snapshot and rename it like the original. This would give us a sort of persistence, but we still needed a mount at the first place. He later suggested completely getting rid of the snapshot, and at the same time coolstar found the same thing supposedly independently. Well, that's what they did at the end.
Let's dive in APFS.kext:

```
205         *(_QWORD *)(*(_QWORD *)(v15 + 160) + 968LL) = v21;
206         if ( v21 )
207         {
208           v36 = *(_QWORD *)(v20 + 112);
209           v37 = "%s:%d: SET root_to_xid - on next boot, volume will root to snapshot \"%s\" w/snap xid %lld\n";
210         }
211         else
212         {
213           v37 = "%s:%d: UNSET root_to_xid - on next boot, volume will root to live file system\n";
214         }
215         sub_FFFFFFF006A4DD98(v37);
216         sub_FFFFFFF006A7C03C(v15, 1LL, 1LL, 0LL, v38);
217         v17 = 0LL;
```

Apparently if no snapshot was found the system would just mount the real /dev/disk0s1s1. And this isn't a bug, it's a sort of feature. Back when snapshots were introduced, restoring one would lead to the same thing, your device would be frozen with that state until the snapshot was gone. A remount would not work. It appears that all his happening now is that a snapshot is pre-installed with iOS. Since during updates a new snapshot has to be made, Apple probably uses this "feature" to switch the filesystems. Delete snapshot, mount device, make snapshot, restore.
The original code of this implementation was written by Pwn20wnd, you can find it here.

Now, what issues would be caused by the lack of r/w? First, code execution:
On *any* kppless jailbreak you **can't** execute binaries in the user partition, you'll get "Operation not permitted" (I've found that if parent process trying to execute has enough permissions then it can execute but that's a different story).

Let's think for a moment about App Store apps. They have to be executed somehow right? And they're obviously located in writable paths, as we can install & uninstall apps. How does that work?

Let's try putting a binary inside the .app and executing it. We get "Killed: 9", that's not the same error as before, is it? The binary is allowed to get executed, it just got killed by something for some reason. Let's try going back a few directories to see what's the smallest path where we can execute. Apps are located in /var/containers/Bundle/Applications/UUID/appname.app, we try / var/containers/Bundle/Applications, then /var/containers/Bundle/ and then /var/containers/.

You'd notice that if we go outside /var/containers/Bundle, we get the old "Operation not permitted" error. In conclusion we can execute as long as we're inside /var/containers/Bundle/. So far, great progress!

Now, how do we solve that "Killed: 9" error? If you remember, Ian Beer's mach_portal exploit didn't have r/w in / but still had a root shell. Usually "Killed: 9" means binary isn't codesigned or has invalid entitlements. If we see what entitlements the binaries used by Ian Beer had (part of Jonathan Levin's pack) you would notice a special entitlement "com.apple.private.security.container-required" set as "false", add that (along with "platform-application" as true, which all kppless jailbreaks, except Meridian require) and binary ran! Glory! It appeared that everything under that directory was applied a container, that seemed to cause these issues.

Now put some binaries inside that path and start a dropbear shell. Part 1 – SSH = done!

Getting around that sandbox part was easy, we now have a shell and everything. I had my own jailbreakd made (check it out here) however launchd wasn't happy with my high-level IPC so at the end I used Electra's jailbreakd (the one that was kind of more stable, but still problematic, at least for me; I could write my own too but this is enough for the moment) along with its pspawn_payload, Tweak Injector & Substitute, patched the paths up (so it would use the symlinks under /var/ which redirected to /var/containers/Bundle and not traditional paths under /), got some test SpringBoard tweak, patched that too and gave it a go. It worked! :)

I later released that as the original rootlessJB and everyone was happy, it didn't cause conflicts with other jailbreaks, no jailbreak detection (almost) and first public jailbreak for iOS 11.3 with tweaking support.

But…

> "*file system sandbox blocked mmap()…*"

Sandbox, you again? Everything was fine for SpringBoard tweaking and all unsandboxed apps but sandboxed stuff would just crash and leave that message on the crash log, forcing me to whitelist only unsandboxed apps from code injection. /var/containers/Bundle is OK with code execution but it acts the same way as /var for code loading. The only supported paths from where you can map new executable memory can be described in a few words: Everything in / that is readable (/System and /usr/lib by default) and **only the app's own container** in /var.

As for Electra which stored tweaks in /Library, initially it had a similar issue but with "stat" instead of "mmap" meaning "this binary isn't allowed to read this file". There was fortunately an entitlement for that com.apple.security.exception.files.absolute-path.read-only, taking an array of paths. Slightly more complicated entitlement-patch this time, the entitlements are cached by sandbox in what are called "sandbox extensions". Create a new extension and there you go. (For more detail check out this write-up by stek29)

I had already applied the sandbox stat() patches and was now looking for a way to bypass mmap() restrictions. Unsandboxing worked but that's quite a terrible idea! Not only bad security-wise but would also mess up app containers and their stuff would be stored in /var/mobile instead of /var/mobile/Containers/Data/Application/APPLICATION_UUID/

I had to think of a more clever idea. Looked for entitlements, no luck, there wasn't any. Had a look at the dyld source code (part of its job is communicating with sandbox and amfi on what we can and can't do) and found the sandbox profile key which had to do with execute mmap restrictions: "file-map-executable", some google-ing and I realized you could add com.apple.security.temporary-exception.sbpl entitlement with a string written in sandbox profile language; it ends up you can't add sandbox profiles via entitlements in iOS, that entitlement is a Mac only thing. So out of luck? :(

I left the whole project behind for much time, Electra was released, people would use that so I had plenty of time to work on it again. I was trying to make sense of how the sandbox slot works and try to find a way to enable mmap() without fully unsandboxing.

After countless of tries I decided to give the sandbox kext a look. Found this function: “mpo_file_check_mmap”, seemed like exactly what I was looking for, looked up some documentation of it and found this.

Especially interesting:

```
@return Return 0 if access is granted, otherwise an appropriate value for
     errno should be returned. Suggested failure: EACCES for label mismatch or
     EPERM for lack of privilege.
```

So if that function returns 0 that means mmap() is allowed. Let’s see *when* that happens

```
1  __int64 __fastcall mpo_file_check_mmap(__int64 a1, __int64 a2, __int64 a3, char a4)
2  {
3    __int64 v4; // x19
4    __int64 v5; // x20
5    __n128 v7; // [xsp+0h] [xbp-140h]
6    char v8; // [xsp+18h] [xbp-128h]
7    int v9; // [xsp+88h] [xbp-B8h]
8    __int64 v10; // [xsp+90h] [xbp-B0h]
9
10   v4 = a1;
11   if ( !(a4 & 4) )
12     return 0LL;
13   if ( **(_DWORD **)(a2 + 40) != 1 )
14     return 0LL;
15   v5 = *(_QWORD *)(a2 + 56);
16   if ( (unsigned int)vnode_isdyldsharedcache(*(_QWORD *)(a2 + 56)) )
17     return 0LL;
18   bzero(&v8, 0x108uLL);
19   v9 = 1;
20   v10 = v5;
21   sub_FFFFFFF006B0ECF0(&v7, v4, 14LL, (__int64)&v8);
22   return v7.n128_u32[0];
23 }
```

Now, a4 is this “int prot” parameter, which would be the same as the prot parameter passed to mmap(); it’s being checked against the “4” flag which is “PROT_EXEC”:

```
#define PROT_EXEC    0x4     /* Page can be executed.  */
```

So yea, no surprises, if we’re not trying to execute, 0 is returned no matter what. However more interesting is the highlighted line. That “vnode_isdyldsharedcache” function. The argument getting passed is a2, so “struct fileglob *fg” defined here. Add 56 to that and get to fg_data which appears to be the file’s vnode. Let’s see what that function does. It’s open source in here. All it does is check against the VSHARED_DYLD flag which as defined in here is 0x000200:

```
#define VSHARED_DYLD    0x000200     /* vnode is a dyld shared cache file */
```

What a useful shortcut from Apple! Libs of dyld_shared_cache automatically bypass mmap() restrictions. Add that flag to the dylib's vnode and bypassed!

rootlessJB – who will bypass mmap for me ?

Now we know a way to bypass mmap() restrictions. However:

- Patch needs to be applied into every dylib in /var
- Vnode changes every time a copy of the file is created thus we need to patch again

A good way to tell if we're validating a new file is amfid. Not only we'll know when we're about to load a dylib but it'll be called each time the vnode changes to update the cached signatures. That sounds like a perfect solution! Almost… Try this and guess what:

- Try to load dylib in /var
- "file system sandbox blocked mmap()"
- Amfid logs out: "Loading dylib from /var!"
- Jailbreakd logs out: "Flags before: … and now: …"
- Try to load again
- Works

So what's going on? Not quite sure but either:

- Amfid is called *after* sandbox
- Amfid returns before jailbreakd finishes and app continues its job

Nevertheless we need to fix that. A quick solution would be hooking mmap(), checking if PROT_EXEC flag is there and trying a few times if failed. But that causes some weird issues and mmap randomly fails so I decided to just hook dlopen() and that seems to work well :)
Code is quite simple:

```
int fixupdylib(char *dylib) {
    #define VSHARED_DYLD 0x000200

    uint64_t vnode = getVnodeAtPath(dylib); // get vnode
    if (!vnode) {
        return -1;
    }
    uint32_t v_flags = KernelRead_32bits(vnode + offsetof_v_flags); // 0x54
    if (v_flags & VSHARED_DYLD) return 0; // check if there already

    KernelWrite_32bits(vnode + offsetof_v_flags, v_flags | VSHARED_DYLD); // add

    v_flags = KernelRead_32bits(vnode + offsetof_v_flags);
    vnode_put(vnode);
    return !(v_flags & VSHARED_DYLD);
}
```

mmap bypass workin' great -> https://twitter.com/Jakeashacks/status/1035536785244336129

One more thing…

We usually would want to install unsandboxed applications. On normal jailbreaks we used to drop them in /Applications, and on iOS 11 jailbreaks we also had to add a special entitlement to unsandbox the app.

/Applications is part of the system partition so not writable but that is not the only place where we can store apps. Previous versions of uicache for example iterated all over the path and added each app to the icon cache, theoretically we could have just changed the path somewhere in /var/containers/Bundle and we'd be good to go, but newer uicache versions use a system API (part of MobileCoreServices.framework) to reload the icon caches (which seemed to have the /Applications path hardcoded). It wasn't feasible to try and patch that path up in memory so I thought about something else instead. Why not just install apps as *user applications*? As long as they do what we need do we care much?

The same framework also provides an API to deal with installing apps, but installd does more codesign checks before app is even installed. We can solve that easily by using AppSync Unified, which does exactly what we need: patch installd. Previous versions of rootlessJB did something hacky, they saw the app's name and if it was "Filza" gave root & unsandboxing, if it was "iSuperSU" they just unsandboxed it. AppSync allows installing apps with arbitrary entitlements so no more need for that! This also means you can't install those apps with Impactor anymore; rather you have to add "no-container: true" & "container-required: false" entitlements, put apps in /var/Apps and run the new uicache (name left like that since people are used to it), which just installs all apps in the directory. Deleting is done like normal App Store apps!

Bonus – The bugs: MPTCP

Now let's start talking about the actual bugs Ian Beer discovered, both of which are buffer overflows. The first one, is the bug in MPTCP, exploited in "multi_path".

The buggy function is 'mptcp_usr_connectx()' defined in this file.

During the code we can notice these size checks:

```
    if (dst->sa_family == AF_INET &&
        dst->sa_len != sizeof(mpte->__mpte_dst_v4)) {
        mptcplog((LOG_ERR, "%s IPv4 dst len %u\n", __func__,
                  dst->sa_len),
                 MPTCP_SOCKET_DBG, MPTCP_LOGLVL_ERR);
        error = EINVAL;
        goto out;
    }

    if (dst->sa_family == AF_INET6 &&
        dst->sa_len != sizeof(mpte->__mpte_dst_v6)) {
        mptcplog((LOG_ERR, "%s IPv6 dst len %u\n", __func__,
                  dst->sa_len),
                 MPTCP_SOCKET_DBG, MPTCP_LOGLVL_ERR);
        error = EINVAL;
        goto out;
    }
```

You can see two checks for different socket families, the first is for IPv4 sockets and the second for IPv6; both these checks make sure the length of the user controlled argument "dst" is equal the size of "mpte->__mpte_dst_vx", where x is either 4 or 6. "mpte" is a "struct mptses" type as defined on the second line of the function: `struct mptses *mpte = NULL;`
You can see the "__mpte_dst_vx" member is part of an union in the "struct mptses" definition (header). All three members of the union are of type "struct sockaddr", so we have the same size regardless of using IPv4 or IPv6. After the checks the code continues as following:

```
if (!(mpte->mpte_flags & MPTE_SVCTYPE_CHECKED)) {
    if (mptcp_entitlement_check(mp_so) < 0) {
        error = EPERM;
        goto out;
    }

    mpte->mpte_flags |= MPTE_SVCTYPE_CHECKED;
}

if ((mp_so->so_state & (SS_ISCONNECTED|SS_ISCONNECTING)) == 0) {
    memcpy(&mpte->mpte_dst, dst, dst->sa_len);
}
```

Immediately after the two 'if' statements the code checks if we have the multi_path entitlement (otherwise an error occurs) and continues with copying data, it made size checks and by the time we got to this part of the code, so we can copy from "dst" (user controlled) to "mpte->mpte_dst", "dst->sa_len" bytes.

```
memcpy(&mpte->mpte_dst, dst, dst->sa_len);
```

Yes, the code did check against size of "mpte->__mpte_dst_vx" and not "mpte_dst", but "mpte_dst" is also part of the same union we talked about earlier, which had three members of type "struct sockaddr", so it really does not matter what you check and where you write on this case.

But, are we sure this is alright? Go back and take a look at the checks again. You might think, if size of "mpte->__mpte_dst_v4" and "mpte->__mpte_dst_v6" is the same, why are two separate checks then? Apple went too specific and logs different errors on both cases, as you can see from the mptcplog() calls, which is probably the reason we have two different checks, and here's the catch: What if we specify a socket family which is neither AF_INET or AF_INET6? Checks are bypassed completely and we can pass an arbitrary size to dst->sa_len! As a result, we can overflow mpte->mpte_dst.

Note: due to sa_len being defined as a unsigned 8 bit integer, the biggest value we can pass is 255 (which is the biggest value you can store in 8 bits: 11111111 in binary).

Apple patched this bug in iOS 11.4 (XNU 4570.61.1) and assigned it CVE-2018-4241. To fix the bug all they did is enforce the socket family to be either AF_INET or AF_INET6, adding this code before size checks were performed:

```
if (dst->sa_family != AF_INET && dst->sa_family != AF_INET6) {
        error = EAFNOSUPPORT;
        goto out;
}
```

Bonus – The bugs: VFS

Ian Beer's second bug involved the getvolattrlist() function, defined here like so:

```
static int getvolattrlist(vfs_context_t ctx, vnode_t vp, struct attrlist *alp,
user_addr_t attributeBuffer, size_t bufferSize, uint64_t options, enum uio_seg
segflg, int is_64bit);
```

The function is big enough, so we'll only take a look at the interesting part. "bufferSize" is a user-controlled argument and it is used to allocate a buffer.

```
    ab.allocated = ulmin(bufferSize, fixedsize + varsize);
    if (ab.allocated > ATTR_MAX_BUFFER) {
        error = ENOMEM;
        VFS_DEBUG(ctx, vp, "ATTRLIST - ERROR: buffer size too large (%d limit %d)",
ab.allocated, ATTR_MAX_BUFFER);
        goto out;
    }
    MALLOC(ab.base, char *, ab.allocated, M_TEMP, M_ZERO | M_WAITOK);
```

The thing that pops out straight away is "ab.allocated > ATTR_MAX_BUFFER", defined as 8192 in attr.h. The allocated buffer can't be bigger than that and nonetheless, even if it could be bigger the buffer would be allocated to contain *that* specific size, so we can't really overflow in here. Let's continue looking through the function. This is the part when data is written on the buffer:

```
    if (return_valid) {
        ab.actual.commonattr |= ATTR_CMN_RETURNED_ATTRS;
        if (pack_invalid) {
            /* Only report the attributes that are valid */
            ab.actual.commonattr &= ab.valid.commonattr;
            ab.actual.volattr &= ab.valid.volattr;
        }
        bcopy(&ab.actual, ab.base + sizeof(uint32_t), sizeof (ab.actual));
    }
```

During this code, we use bcopy to copy from ab.actual to ab.base + 4 (size of uint32_t). Let's start analyzing:

"ab" is defined as "struct _attrlist_buf" in the same file. ab.base is a char pointer (reference) and ab.actual is a type of "attribute_set_t", which as defined in here is a "struct attribute_set", consisting of 5 members, each of type "attrgroup_t" which is actually a 32 bit (4 bytes) unsigned integer. 4 times 5 = 20 bytes, which is the size of the whole struct.

Bonus – The bugs: VFS

So to recap, ab.base was allocated with our specified bufferSize (with a limit up to 8192), ab.actual is a 20 byte struct and we're copying 20 bytes from ab.actual at the address 4 bytes after ab.base. What's wrong here? The size checks make sure ab.base is not too big, but nowhere there's a check if it is too small, what if our specified buffer size (-4 since we're writing 4 bytes *after* ab.base) is smaller than 20 bytes? If we pass 16 as the size, 16 - 4 = 12, we're copying 20 bytes, so 20 - 12 = 8 more bytes than what the buffer can hold! We can now overflow.

Apple patched this bug in iOS 11.4 (XNU 4570.61.1) and assigned it CVE-2018-4243. Before allocating the buffer they make necessary size checks now, the buffer size must be at least size of uint32_t + size of attribute_set_t which is 4 + 20 = 24 bytes

```
    if (return_valid && (ab.allocated < (ssize_t)(sizeof(uint32_t) +
sizeof(attribute_set_t))) && !(options & FSOPT_REPORT_FULLSIZE)) {
        uint32_t num_bytes_valid = sizeof(uint32_t);
        /*
         * Not enough to return anything and we don't have to report
         * how much space is needed. Get out now.
         * N.B. - We have only been called after having verified that
         * attributeBuffer is at least sizeof(uint32_t);
         */
        if (UIO_SEG_IS_USER_SPACE(segflg)) {
            error = copyout(&num_bytes_valid,
                            CAST_USER_ADDR_T(attributeBuffer), num_bytes_valid);
        } else {
            bcopy(&num_bytes_valid, (void *)attributeBuffer,
                  (size_t)num_bytes_valid);
        }
        goto out;
    }
```

Huge thanks to:

Ian Beer for his awesome exploits and tricky-to-find bugs

stek29 for his awesome work

Jonathan Levin for his awesome write-ups and tools

IBSparkes for being an awesome man

To check out:

jelbrekLib, an open-source jailbreak library: https://github.com/jakeajames/jelbrekLib

kernelSymbolFinder, an open-source on-device kernel symbol finder, (no jailbreak required; no iOS 12 support (yet)!): https://github.com/jakeajames/kernelSymbolFinder

iSuperSU, basic SuperSU-style app for iOS to easily manage privileges of your apps: https://github.com/jakeajames/iSuperSU