dyld closure exploit

dyld has an interesting feature called "dyld closures", which are used to speed-up App launches on iOS.
When launching an App for the first time, dyld links the executable and then writes the result to a .closure cache file, inside the App's HOME Folder ($HOME/Library/Caches/com.apple.dyld/<appname>.closure)
On subsequent starts of the App, the executable is linked using this cache file which already contains all the information needed to quickly link it.
However, by creating a crafted closure it is possible to change how the executable is linked, therefore making it possible to inject code.
Closures have many interesting features:

- Arbitrary memory in the App can be changed to point into a library, the shared cache or somewhere else inside the main executable.
- Additionaly, arbitrary data can be written into memory, by linking to a "static address".
- It is also possible to sign pointers, because this is a required feature on arm64e.
- The closure also contains the entry point for the executable (which can even be in a library instead of the executable).

Closures can also include information for other libraries, which will then be loaded into the process.
These libraries can also be modified.

Exploitation

iOS

To exploit this vulnerability, an interesting App has to be found first.
On iOS, every App which has entitlements that force it to not run inside the sandbox is interesting.
There are quite a few, but I've (arbitrarily) chosen Spotlight (/Applications/Spotlight.app/Spotlight)
However, how can such an App be forced to start with a malicious closure?
The answer is: Just install an App which has it's executable replaced with the entitled App's executable.
This, however, requires a second bug in order to bypass some checks during installation (see the next part for this vulnerability).
Now when the App is run, the entitled executable is run and loads the modified closure.
Of course it is necessary to place such a modified closure inside the App's HOME Folder first.
This can be done by first uploading a regular App, which creates the closure.
Afterwards the App is then updated to a new version containing the entitled executable.

However, there are a few checks that have to be bypassed first:

1. Check for closures inside the dyld shared cache

Before dyld attempts to load the closure inside the App's HOME folder, it checks if a valid closure is inside the dyld shared cache.
Code (`dyld2.cpp`):

```
1         // check for closure in cache first
```

```
        if ( sSharedCacheLoadInfo.loadAddress != nullptr ) {
            mainClosure = sSharedCacheLoadInfo.loadAddress->findClosure(sExecPath);
            if ( gLinkContext.verboseWarnings && (mainClosure != nullptr) )
                dyld::log("dyld: found closure %p (size=%lu) in dyld shared cache\n",
mainClosure, mainClosure->size());
            if ( mainClosure != nullptr )
                sLaunchModeUsed |= DYLD_LAUNCH_MODE_CLOSURE_FROM_OS;
        }

        // findClosure method:
        const dyld3::closure::LaunchClosure* DyldSharedCache::findClosure(const char*
executablePath) const
        {
            const dyld_cache_mapping_info* mappings = (dyld_cache_mapping_info*)((char*)this
+ header.mappingOffset);
            uintptr_t      slide                = (uintptr_t)this - (uintptr_t)
(mappings[0].address);
            const uint8_t* executableTrieStart  = (uint8_t*)(this-
>header.progClosuresTrieAddr + slide);
            const uint8_t* executableTrieEnd    = executableTrieStart + this-
>header.progClosuresTrieSize;
            const uint8_t* closuresStart        = (uint8_t*)(this->header.progClosuresAddr +
slide);

            Diagnostics diag;
            const uint8_t* imageNode = dyld3::MachOLoaded::trieWalk(diag,
executableTrieStart, executableTrieEnd, executablePath);
            if ( (imageNode == NULL) && (strncmp(executablePath, "/System/", 8) == 0) ) {
                // anything in /System/ should have a closure.  Perhaps it was launched via
symlink path
                char realPath[PATH_MAX];
                if ( realpath(executablePath, realPath) != NULL )
                    imageNode = dyld3::MachOLoaded::trieWalk(diag, executableTrieStart,
executableTrieEnd, realPath);
            }
            if ( imageNode != NULL ) {
                uint32_t closureOffset = (uint32_t)dyld3::MachOFile::read_uleb128(diag,
imageNode, executableTrieEnd);
                if ( closureOffset < this->header.progClosuresSize )
                    return (dyld3::closure::LaunchClosure*)((uint8_t*)closuresStart +
closureOffset);
            }

            return nullptr;
        }
```

This check is easily bypassed, because it will not find a closure if the executable is not at it's standard path. Additionally, if the executable was launched from a path that doesn't start with "/System/", it is sufficient to use a symlink to the real executable.
In case a closure is found, but the closure is not for this executable (checked using the CDHash), it will be discarded, also leading to the closure inside the HOME folder being used, because of the following code:

```
        // We only want to try build a closure at runtime if its an iOS third party binary,
or a macOS binary from the shared cache
        bool allowClosureRebuilds = false;
        if ( sClosureMode == ClosureMode::On ) {
```

```
            allowClosureRebuilds = true;
        } else if ( (sClosureMode == ClosureMode::PreBuiltOnly) && (mainClosure != nullptr)
) {
            allowClosureRebuilds = true;
        }
```

2. Check that the boot hash matches

Before loading a closure from the HOME folder, dyld check that a special xattr on the closure matches the current boot hash, which is regenerated on each boot. Therefore, normally closures wouldn't be reused after a reboot and instead be regenerated.
However, there is a bug in `buildLaunchClosure`:

```
        // make new file
        // ... create temporary closure path using PID (closurePathTemp)
#if TARGET_OS_OSX
        int fd = dyld3::open(closurePathTemp, O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
#else
        int fd = ::open_dprotected_np(closurePathTemp, O_WRONLY|O_CREAT, PROTECTION_CLASS_D,
0, S_IRUSR|S_IWUSR);
#endif
        if ( fd != -1 ) {
            ::ftruncate(fd, result->size()); // Truncate temporary file
            ::write(fd, result, result->size()); // Write closure to temporary file
            ::fsetxattr(fd, DYLD_CLOSURE_XATTR_NAME, bootToken.begin(), bootToken.count(),
0, 0); // Set magic xattr on it
            ::fchmod(fd, S_IRUSR);
            ::close(fd);
            ::rename(closurePathTemp, closurePath); // Rename temporary file to real closure
*without checking if rename succeded*
            // free built closure and mmap file() to reduce dirty memory
            result->deallocate();
            result = mapClosureFile(closurePath); // Map real closure
            sLaunchModeUsed |= DYLD_LAUNCH_MODE_CLOSURE_SAVED_TO_FILE;
        }
```

First the newly generated closure is written to a temporary file. Then the new file is renamed to the .closure file. Afterwards, the closure will be read again from the .closure file. However, no checks are performed to check if the rename actually succeded.
By e.g. setting the UF_IMMUTABLE bit on the closure (using chflags), dyld will be unable to replace it and when reading the closure again, the modified one will be used instead.

macOS

Exploitation on macOS is similar to iOS, with the difference that by default macOS won't use closures, except those inside the dyld shared cache.

To bypass this restriction, dyld has to find a valid closure inside the shared cache, but this closure must not match the current program.
In this case, dyld will fall back to loading a cached closure (from $HOME/Library/Caches/com.apple.dyld/<appname>.closure).
Forcing dyld to find a non-matching closure inside the shared cache can be done by performing these steps:

1. Symlink the target executable (e.g. setuid+entitled) to /tmp/target
2. Launch the executable via `posix_spawn` using the path /System/../tmp/target, making sure the

`POSIX_SPAWN_START_SUSPENDED` flag is set
3. Replace /tmp/target with a symlink to /bin/bash (or something else that has a closure inside the shared cache)
4. Send SIGCONT to the executable to continue it
5. Dyld will see that the executable path starts with /System/ and resolve the symlink to find the closure inside the shared cache
6. The closure will not match the executable and dyld will try to load a cached closure from $HOME/Library/Caches/com.apple.dyld/<appname>.closure

Closure generation

Please see the Section `Jailbreak - dyld closure exploit` for technical details on how closures are generated.

Fix

First of all, disable the use of cached closures on macOS if the closure in the dyld shared cache is "outdated". Remove this in `dyld2.cpp`:

```
        else if ( (sClosureMode == ClosureMode::PreBuiltOnly) && (mainClosure != nullptr) )
  {
            allowClosureRebuilds = true;
        }
```

Additionally, disable the use of cached closures if the current process is a platform application. This should mitigate this issue on iOS.
For example, add this to `dyld2.cpp`:

```
        // allowClosureRebuilds already exist
        bool allowClosureRebuilds = false;
        // New code starts here
        if (sJustBuildClosure) {
            // If sJustBuildClosure is set, always allow rebuilding
            // In this case dyld will exit before using the closure
            allowClosureRebuilds = true;
        } else if (sClosureMode == ClosureMode::On) {
            // Only allow closure rebuilds if this is not a platform binary
            uint32_t flags = CS_PLATFORM_BINARY;
            if (csops(0, CS_OPS_STATUS, &flags, sizeof(flags)) != -1) {
                if ((flags & CS_PLATFORM_BINARY) == 0) {
                    allowClosureRebuilds = true;
                }
            }
        }
```

DriverKit Kernel Exploit

In macOS Catalina, System Extensions have been introduced as a safer alternative to kernel extensions:
By running in user space, system extensions can't compromise the security or stability of macOS.

One of the System Extension frameworks is DriverKit, which can be used to implement drivers in userspace. As the XNU kernel was not designed for this and because DriverKit can be used to directly talk to hardware, I decided to look for vulnerabilities in this framework.

iOS Note

Interestingly, DriverKit is completely present on iOS although it is seemingly not used by anything.
The only problem is that there is no kernelmanagerd, so it is impossible to load a DriverKit driver through "normal" means.
However, it is possible to replicate kernelmanagerd enough to be able to load a DriverKit driver.

Bug

There is a Bug in the DriverKit method `CreateMemoryDescriptorFromClient` of `IOUserClient`:

```
1 kern_return_t
2 IOUserClient::CreateMemoryDescriptorFromClient_Impl(
3     uint64_t memoryDescriptorCreateOptions,
4     uint32_t segmentsCount,
5     const IOAddressSegment segments[32],
6     IOMemoryDescriptor ** memory)
7 {
8     IOReturn             ret;
9     IOMemoryDescriptor * iomd;
10    IOOptionBits         mdOptions;
11    IOUserUserClient   * me;
12    IOAddressRange     * ranges;
13
14    me = OSDynamicCast(IOUserUserClient, this);
15    if (!me) {
16        return kIOReturnBadArgument;
17    }
18
19    mdOptions = 0;
20    if (kIOMemoryDirectionOut & memoryDescriptorCreateOptions) {
21        mdOptions |= kIODirectionOut;
22    }
23    if (kIOMemoryDirectionIn & memoryDescriptorCreateOptions) {
24        mdOptions |= kIODirectionIn;
25    }
26    if (!(kIOMemoryDisableCopyOnWrite & memoryDescriptorCreateOptions)) {
27        mdOptions |= kIOMemoryMapCopyOnWrite;
28    }
```

```
29
30      static_assert(sizeof(IOAddressRange) == sizeof(IOAddressSegment));
31      ranges = __DECONST(IOAddressRange *, &segments[0]);
32
33      iomd = IOMemoryDescriptor::withAddressRanges(
34          ranges, segmentsCount,
35          mdOptions, me->fTask);
36
37      if (iomd) {
38          ret = kIOReturnSuccess;
39          *memory = iomd;
40      } else {
41          ret = kIOReturnNoMemory;
42          *memory = NULL;
43      }
44
45      return ret;
46  }
```

The Bug is in this part:

```
1       iomd = IOMemoryDescriptor::withAddressRanges(
2           ranges, segmentsCount,
3           mdOptions, me->fTask);
```

There is no check that me->fTask is not NULL!

This actually a very serious vulnerability, because the Documentation for `IOMemoryDescriptor::withAddressRanges` tells us this:
@param task The task each of the virtual ranges are mapped into.
Note that unlike IOMemoryDescriptor::withAddress(), kernel_task memory must be explicitly prepared when passed to this api.
The task argument may be NULL to specify memory by physical address.

By creating a new IOUserClient without initializing it with a task (which is completely normal as the task will only be set later by the kernel), it is possible to create IOMemoryDescriptors for arbitrary physical memory! This descriptor can then be used to read/write the memory or map it into the current process, which is enough to compromise the whole kernel.

Fix

The fix is easy: Just check that `me->fTask` is not NULL.
For example:

```
1  kern_return_t
2  IOUserClient::CreateMemoryDescriptorFromClient_Impl(
3      uint64_t memoryDescriptorCreateOptions,
4      uint32_t segmentsCount,
5      const IOAddressSegment segments[32],
6      IOMemoryDescriptor ** memory)
7  {
8      IOReturn             ret;
9      IOMemoryDescriptor * iomd;
10     IOOptionBits         mdOptions;
11     IOUserUserClient   * me;
```

```
12      IOAddressRange      * ranges;
13
14      me = OSDynamicCast(IOUserUserClient, this);
15      if (!me) {
16          return kIOReturnBadArgument;
17      }
18
19      if (me->fTask == NULL) {
20          return kIOReturnNotReady; // me->fTask not yet set so kIOReturnNotReady seems like a
    good choice
21      }
22
23      // ...
```

Kernel PAC bypass

In order to call functions in the kernel, a PAC bypass is required on iPhone Xs and later.
Additionally, the PAC bypass must not use the hardware vulnerability discovered by Project Zero because it has been patched on the iPhone 12.
I've decided to target exception frames saved to the stack, because they seemed like an obvious way to gain code execution.

Observations

Looking through the code it appeared that the exception frame is signed after being written to the stack.
However, only "important" registers (x16, x17, lr, pc, cpsr) are used to generate the signature.
Unfortunately, without controlling these registers it becomes really hard to redirect control flow.
I've looked into other ways, like setting breakpoints or watchpoints to get a predictable exception frame (e.g. when signing some pointers), but the kernel will always panic when setting breakpoints or watchpoints that can be triggered inside the kernel.
Additionally, on the iPhone XS and later it appears that most external debugging registers simply don't exist or have to be enable first somehow. If those were enabled, the physical address mapping primitive from the kernel exploit would have been really interesting.
After spending some time investigating more possibilities, I returned to the exception state signing code and noticed that it is also used to sign user thread states (using the same key).
I therefore wondered if it would be possible to abuse this fact to resign kernel states.

Bug

Looking into `thread_state64_to_saved_state` (used by `machine_thread_set_state`), I found out that the code never touches important bits in the saved cpsr register.
These important bits include the mode bits, specifying whether or not the thread is running in kernel mode.
Therefore, it is indeed possible to resign kernel states: Just change the kernel stack of a thread (thread A) to the user state of another thread (thread B) and then force an exception in the kernel (in thread A), making sure the exception will be written exactly to the user state of thread B. Thread B will now have a kernel state which can be modified through `thread_set_state` and once thread B is resumed, it will run in kernel mode!

Fix

You could fix `thread_state64_to_saved_state` like this:

```
1 void
2 thread_state64_to_saved_state(const arm_thread_state64_t * ts64,
3     arm_saved_state_t *          saved_state)
4 {
5     uint32_t i;
6 #if __has_feature(ptrauth_calls)
7     boolean_t intr = ml_set_interrupts_enabled(FALSE);
```

```
#endif /* __has_feature(ptrauth_calls) */

    assert(is_saved_state64(saved_state));

#if __has_feature(ptrauth_calls)
    MANIPULATE_SIGNED_THREAD_STATE(saved_state,
        "and    w2, w2, %w[not_psr64_user_mask] \n"
        "mov    w6, %w[cpsr]                    \n"
        "and    w6, w6, %w[psr64_user_mask]     \n"
        "orr    w2, w2, w6                      \n"
        "str    w2, [x0, %[SS64_CPSR]]          \n",
        [cpsr] "r"(ts64->cpsr),
        [psr64_user_mask] "i"(PSR64_USER_MASK),
        [not_psr64_user_mask] "i"(PSR64_USER_KEEP_MASK)
        );

    // ...

// Put this definition into proc_reg.h:
#define PSR64_USER_KEEP_MASK (~(PSR64_USER_MASK | PSR64_MODE_MASK))
```

PPL bypass

There is a PPL protected function called `pmap_enter_options_internal` (in xnu/osfmk/arm/pmap.c) which is used to map a page into a pmap (a pmap is essentially the pagetable for a process; pmaps are protected by PPL). This function takes a physical address (pa), a virtual address (va) plus some options and will then map the va to the pa in the pmap.
However, before doing so, some checks are performed on the pa to ensure that it is not part of PPL, because otherwise it would be trivial to bypass PPL (e.g. if a process maps it's own pmap it could access whatever memory it likes by directly changing the pagetable entries).

Let's look through all the checks performed by the function (uneccesary code has been removed).

1st check right at the beginning:

```
1 if ((v) & pt_attr_leaf_offmask(pt_attr)) {
2     panic("pmap_enter_options() pmap %p v 0x%llx\n", pmap, (uint64_t)v);
3 }
4
5 if ((pa) & pt_attr_leaf_offmask(pt_attr)) {
6     panic("pmap_enter_options() pmap %p pa 0x%llx\n", pmap, (uint64_t)pa);
7 }
```

This essentially just checks that both the va and pa are page-aligned.
As pmap only operates on whole pages, this check is necessary.

2nd check:

```
1 if (pa_valid(pa)) {
2     // ...
3
4     /* The regular old kernel is not allowed to remap PPL pages. */
5     if (__improbable(pa_test_monitor(pa))) {
6         panic("%s: page belongs to PPL, "
7             "pmap=%p, v=0x%llx, pa=%p, prot=0x%x, fault_type=0x%x, flags=0x%x, wired=%u,
  options=0x%x",
8             __FUNCTION__,
9             pmap, v, (void*)pa, prot, fault_type, flags, wired, options);
10    }
11
12    if (__improbable(pvh_get_flags(pai_to_pvh(pai)) & PVH_FLAG_LOCKDOWN)) {
13        panic("%s: page locked down, "
14            "pmap=%p, v=0x%llx, pa=%p, prot=0x%x, fault_type=0x%x, flags=0x%x, wired=%u,
  options=0x%x",
15            __FUNCTION__,
16            pmap, v, (void *)pa, prot, fault_type, flags, wired, options);
17    }
18
19    // ...
20 } else {
```

```
21     // No checks whatsoever
22 }
```

This check is interesting: If the pa is "valid", the function will check if the pa belongs to PPL or is "locked down".
But how does the function determine if a pa is valid?
Definition of `pa_valid`:

```
1 #define pa_valid(x) ((x) >= vm_first_phys && (x) < vm_last_phys)
```

So `pa_valid` only checks that the pa is between `vm_first_phys` and `vm_last_phys`.
Every physical address outside this range is considered not valid.

Now what would happen if we specify an invalid pa?
Note that the pa is 64 bits, but aarch64 only supports 48 bit physical addresses.
What would happen if one of the high bits is set? E.g. bit 63?
Such an address is definitly invalid.

Let's see how the page table entry is constructed:

```
1 #define ARM_PTE_PAGE_MASK 0x0000FFFFFFFFF000ULL /* output address mask for page */
2 #define pa_to_pte(a)      ((a) & ARM_PTE_PAGE_MASK)
3
4 pte = pa_to_pte(pa) | ARM_PTE_TYPE;
```

As expected, only the low 48 bits (minus the first 12 bits, for alignment) of the physical address are used to construct the page table entry.

Looking through the source code again, it didn't seem as if anything would check that the high bits of the pa are unset, so I decided to just try it out.
And indeed, by using a PPL protected pa and setting bit 63, all checks can be bypassed.
Because bit 63 will be ignored when creating the page table entry, the PPL protected address will be mapped into the pmap, therefore completely bypassing PPL.

Exploitation

Exploiting this vulnerability is really easy:
Just call `pmap_enter_options_addr` (a wrapper around `pmap_enter_options_internal`) with a PPL protected pa, but make sure to set bit 63 on the pa first.
The PPL protected address will now be mapped into your process.

Warning: Never change the mapping for this va (e.g. by unmapping or remapping), because this *will* cause a kernel panic.
Exiting, however, is safe, as no checks will be performed when destroying a pmap, only when directly unmapping an address.

Fix

Add an aditional check to the beginning of the function, like this:

```
1     if (__improbable(pa & ARM_PTE_PAGE_MASK)) {
2         panic("%s: pa is invalid, "
3                 "pmap=%p, v=0x%llx, pa=%p, prot=0x%x, fault_type=0x%x, flags=0x%x, wired=%u,
  options=0x%x",
4                 __FUNCTION__,
```

```
5                 pmap, v, (void*)pa, prot, fault_type, flags, wired, options);
6     }
```

Jailbreak

High level overview

These are the steps performed by the Jailbreak:

1. Install first App and click on the Setup button
2. Files for the dyld closure exploit are generated and written to $HOME/Caches/com.apple.dyld
   -> Closures are generated for /Applications/Spotlight.app (not sandboxed), keybagd (has an entitlement [`com.apple.private.persona-mgmt`] to launch other Applications as root via `posix_spawnattr_set_persona_uid_np`), installd (to set the executable bit on the kernel exploit), ReportCrash (has `task_for_pid-allow`, used to patch amfid so the kernel exploit can be launched)
3. Install second App and run it
4. The second App has it's executable replaced by the one of Spotlight (Spotlight is unsandboxed) and the dyld closure exploit will be triggered, granting code execution in Spotlight
5. Spotlight launches keybagd (triggering the dyld closure exploit again)
6. keybagd launches installd as root, which will then chmod +x the kernel exploit (dyld exploit again)
7. keybagd launches ReportCrash which will then patch amfid so the kernel exploit can be launched (dyld exploit one more time)
8. ReportCrash patches amfid and launches jailbreakd
9. jailbreakd is now marked as having a valid signature and terminates
10. Spotlight (which is still running) exec's into jailbeakd (so the UI can be shown)
11. After tapping on the Jailbreak button, jailbreakd launches a copy of itself as root (via the `com.apple.private.persona-mgmt` entitlement)
12. The root copy of jailbreakd exploits the kernel using the DriverKit exploit to gain arbitrary read/write
13. The Kernel PAC bypass is set up using the arbitrary read/write
14. A custom trustcache is injected using the PPL bypass
15. The root file system is mounted writable and the apfs snapshot is renamed
16. `/System/Library/PrivateFrameworks/CoreAnalytics.framework/Support/analyticsd` is renamed to `analyticsd.back` and `/usr/libexec/keybagd` is copied to `/System/Library/PrivateFrameworks/CoreAnalytics.framework/Support/analyticsd` so that the dyld closure exploit is triggered inside keybagd
17. In /etc/passwd and /etc/master.passwd, the UID and GID for _analyticsd is changed and a new _nanalyticsd user is created, having the UID/GID _analyticsd originally had
18. Additionally, the HOME directory for _analyticsd is changed to `/private/var/mobile/Containers/Data/Fugu14Untether` (with `/private/var/mobile/Containers/Data/Fugu14Untether/Library` being a symlink to `/private/var/Fugu14UntetherDYLD`) so the dyld closure exploit can be triggered
19. Closures are generated for the untether and written to `/private/var/Fugu14UntetherDYLD/Caches/com.apple.dyld/`
20. Finally, the device is rebooted

Untether

When the device boots, the following steps are performed:

1. launchd executes `/System/Library/PrivateFrameworks/CoreAnalytics.framework/Support/analyticsd`, which has been replaced by `/usr/libexec/keybagd`
2. dyld sees that `$HOME` points to `/private/var/mobile/Containers/Data/Fugu14Untether`, which is a valid directory for dyld closures
3. dyld proceeds to load the closure for keybagd in `$HOME/Library/Caches/com.apple.dyld` (with `$HOME/Library` being a symlink to `/private/var/Fugu14UntetherDYLD`)
4. keybagd launches ReportCrash as root (see step seven above)
5. ReportCrash patches amfid and launches `/.Fugu14_Untether/jailbreakd`
6. jailbreakd exploits the Kernel via the DriverKit exploit
7. jailbreakd sets the PAC and PPL bypass up and injects a custom trust cache (containing the signatures for various tools)
8. Finally, the root file system is remounted r/w
9. Additionally, jailbreakd now listens on port 1337 (localhost only) for commands (iDownload)
   -> To get a shell, create a tunnel to the device (`iproxy 1337 1337`) and then connect via netcat (`nc localhost 1337`).
   -> You should now be connected to iDownload, type `bash` to get a bash.

Jailbreak - dyld closure exploit

The dyld closure vulnerability is exploited multiple times to escape the sandbox, become root and patch amfid. Therefore, an easy to use API has been developed, which makes generating the required closures easy.

API description

Writing payloads

In order to generate a new closure, all that needs to be done is to subclass `PwnClosure` (or `GenericJSClosure` if a JavaScript runtime is needed).
Within the new class, the method `generatePayload` has to be overridden. This method generates the actual payload.
Within the payload, it is possible to call arbitrary ObjectiveC methods via the `invoke` method as well as C functions via `callCFunc`.
Return values can be used as arguments to other function/method calls or stored to memory.

For more complex payloads, a JavaScript runtime is provided by the `GenericJSClosure` class.
Two helper files provide convinient JavaScript functions e.g. to support calling C functions.
To use the JavaScript runtime, first initialize it via `initJSRuntime`, pasing the paths of the helper JavaScript files.
Afterwards, `runJSFile` may be used to run JavaScript files.

Generating the closure

In order to generate an actual closure, create an instance of the payload class and call it's `getClosure` method to get a LaunchClosure object.
This LaunchClosure object may then be written to a file by first converting it into a Data object (via the `emit` method) and then writing it to a file via `write(to:)`.

Closure generation

Note: This is a technical description on how the closure is generated from a payload object

Generating a bare-bones closure

The following steps are performed by `createExploitClosure` in `arm/Shared/ClosurePwn/Sources/ClosurePwn/DyldClosure/ClosureBuilder.swift`.
First, a LaunchClosure is prepared, containing the following infos:

- The list of images in this closure
- UUID of the currently used dyld shared cache
- Flags
- Index of libSystem inside the image list
- libDyld entry point (reference to the dyld3::entryVectorForDyld function)

- Index of the top image (i.e. the main image inside the image list)
- Entrypoint of the Application (can point into any image or the shared cache)
- Empty list of missing files

Additionally, two images are added two the image list: a main image and a pwn image.
The main image only contains enough information so dyld won't complain.
Additionally, it declares that it depends on the pwn image as well as a data image (which will be added later)
The pwn image contains the path and other information of a dynamic library which is forced to be loaded into the process.
This is done so that the pwn image can be used as an entry point (and therefore it doesn't matter what the actual main image is).

Filling the closure

The `PwnClosure` class (`arm/Shared/ClosurePwn/Sources/ClosurePwn/PwnClosure.swift`) uses the `createExploitClosure` to first create a bare-bones closure and then fills it.
After the bare-bones closure is created, another image (the data image) is added to it, which is only used to have a writeable memory region inside the target process.
By abusing the linking abilities of closures, data is later written by the closure to the target image, which is then used to start a SLOP payload.
On iOS 14.5, ISA pointer signing has been added to prevent SLOP attacks.
However, closures have the ability to sign arbitrary pointers, therefore bypassing this mitigation.

In order to write memory, the following has to be done:

1. The data to be written (which may be pointers into other images, pointers into the dyld shared cache or 64 bit values) is split into two arrays: Data that needs to be PAC signed and data that doesn't
2. A new BindFixups list is created for the data image
3. The PAC signed data is changed into the chained fixups format and appended to the bind fixups
4. All non-signed data is appended to the bind fixups
5. A fake chained fixups struct (`dyld_chained_starts_in_image/dyld_chained_starts_in_segment`) is created and appended to the bind fixups
6. The image is marked as having chained fixups

When dyld now processes the data image from the closure, the following happens:

1. dyld processes the bind fixups, which causes a fake chained fixups structure to be written inside the data image as well as all unsigned data
2. dyld processes the chained fixups (which are read from the data image), links them according to the information inside the closure and signs them using the specified key and context (key and context were written in the previous step)

Afterwards, the data image now contains all the injected data from the closure, which is additionally signed where needed.
All that's left is to start a SLOP chain using this data.

Starting SLOP chain

In order to start the SLOP chain, the entrypoint is set to code inside the `/usr/lib/libMTLCapture.dylib` library (the pwn image).
This library was chosen because it was written in ObjectiveC and is not part of the dyld shared cache.
Specifically, the following code is required to start the SLOP chain:

```
id res = objc_msgSend(SomeClass, SomeSelector);
objc_retainAutoreleasedReturnValue(res);
```

The linking information of the pwn image is changed so that `SomeClass` points to the root `NSInvocation` object inside the data image, `SomeSelector` is set to -1, `objc_msgSend` is changed to a ROP gadget (see below) and `objc_retainAutoreleasedReturnValue` is changed to the `-[NSInvocation invoke]` method.

In order to be able to use `NSInvocation` objects, a mitigation has to be bypassed: `NSInvocation` objects contain a magic value, which must be the same as in the `magicCookie.oValue` global variable.
This variable, however, is initialized to a random value.
There are two ways to bypass this mitigation: Either change `magicCookie.oValue` directly (which doesn't work in this case as it is not initialized yet) or prevent the initialization of `magicCookie.oValue` by writing -1 to `magicCookie.oGuard`.
The following ROP gadget was chosen:

```
ldr xA, [x0, #x0Offset] // x0Offset >= 0x20 && (x0Offset < 0x30 || x0Offset >= 0x48)
str x1, [xA, #xAOffset]
ret
```

As x0 will be set to the root `NSInvocation` object, the gadget must be chosen so that it loads an address from x0 at some offset which is not used by the root `NSInvocation` object and then stores x1 (set to -1) to this address (optionally at some offset of this address because there is no gadget with offset 0).
Additionally, xA must not be x0 (the root NSInvocation object *must* be returned) or x1.
This way, after the ROP gadget is run `objc_retainAutoreleasedReturnValue` - which is redirected to `-[NSInvocation invoke]` - is called with the first parameter set to the root `NSInvocation` object.

Jailbreak - DriverKit exploit

The DriverKit exploit is used to gain arbitrary memory r/w inside the kernel.
This required reimplementing the complete DriverKit RPC mechanism as well as some parts of kernelmanagerd (which is not available on iOS).

"Loading" a DriverKit driver

Note: The code for this part can be found in `arm/Shared/KernelExploit/Sources/KernelExploit/codelessKext.swift` (in `getDKCheckinData`)

In order to be able to use the DriverKit kernel APIs, a `checkin Token` is required, which is created by the kernel and normally sent to `kernelmanagerd` (which doesn't exist on iOS) when the kernel requests a DriverKit driver to be loaded.
Therefore, it is necessary to register a DriverKit driver with the kernel, force the kernel to send a load request for it and then get the `checkin Token`.
To do this, a `Codeless Kernel Extension` has to be "loaded", which is essentially just a stripped-down version of the `Info.plist` of a DriverKit driver.
This `Codeless Kernel Extension` only consists of `IOKit Personalities`, which describe the conditions for a DriverKit driver to be loaded.
By making sure to always fullfill the conditions, the kernel will then immediately send a launch request and a `checkin Token`, which can then be used to access the DriverKit APIs.

Exploiting the kernel

Note: The code for this part can be found in `arm/Shared/KernelExploit/Sources/KernelExploit/DK.swift` (in `dkLaunchExploit`)

After obtaining a checkin token, the following steps are performed:

1. A IOUserServer object is created using the checkin token
2. A root dispatch queue is created so that messages from the kernel can be received
3. The IOUserServer object is registered with the kernel
4. The kernel sends a start request for the IOUserServer object via the root dispatch queue
5. A new user client is created but not attached to a task
6. The user client can now be used to get memory descriptors for arbitrary physical memory (`IOUserClient.CreateMemoryDescriptorFromClient`)

In order to read/write memory, a `IODMACommand` object is created.
This `IODMACommand` object can then be used to read from or write to a memory descriptor.