A Worm's Look Inside: Apple's Sandboxing security measures on macOS & iOS

Osama Alhour - https://github.com/NSAntoine
August 17, 2024

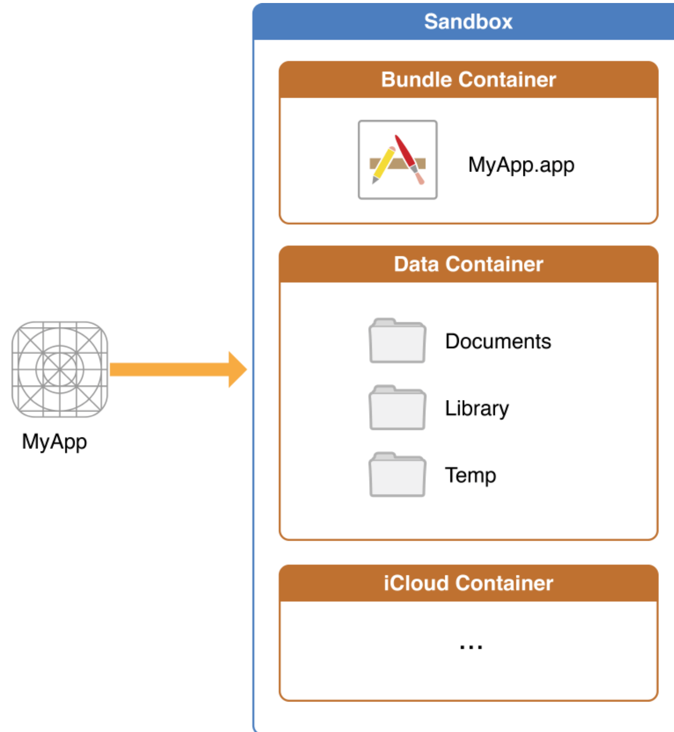
ABSTRACT
This paper explores the security model, internal implementation, and interface of Apple's notorious Sandbox system on iOS and macOS, meant to protect users and their devices from harm by bad actors through a seemingly simple yet intricate-in-practice system of isolating apps and restricting their capabilities in order to prevent as much damage as possible from being done, a practice known as "Mandatory Access Control" [1] which has been adopted by other vendors such as "SELinux" [2], while Apple's own implementation is derived from an existing one by the name of "TrustedBSD" [3]. This paper will also delve into how the Sandbox system works together with other system components.

1. INTRODUCTION
Apple's core implementation of the Sandbox resides within a Kernel Extension (colloquially known as a "kext", modules which add onto the functionality of the kernel, similar to Kernel modules on Linux) by the name of Sandbox.kext, meaning it runs from Kernel-land, an extremely important implementation detail as it to be separated from userland where the checks and balances that are to be enforced by Sandbox.kext could just be hooked and disabled. The simple idea of the Sandbox is to restrict the ability of an application from reading, writing, and interacting with other applications and their files on the device at all to prevent bad actors from exploiting attack vectors which could easily allow them to steal user data, maliciously tamper with user files, etc. if applications were just allowed to roam around the filesystem and read / write anywhere. Apple implements a system where every application is given its own folder that only the application is allowed to read and write to, known as a "container"[1], as well as imposing other restrictions such as filtering system calls. The location of an application's container as well as when it's created depends on platform. On macOS, a sandboxed application's container is created at ~/Library/Containers/{ApplicationID}, i.e., the container for Microsoft Word's would be in ~/Library/Containers/com.microsoft.Word, and the container would be created upon the application opening, if it doesn't already exist.

On iOS, the container situation is a little more complicated. Static application data such as the Application Bundle (where resources like assets bundled with the app, the actual application binary, and libraries / frameworks used in the application, stuff that is guaranteed to never change until the application is updated) is placed in

[1] Containers are isolated to only the app that owns them, however, App Groups exist where multiple apps can access a container, as long as those apps are signed by the same Team ID, a unique identifier assigned to each development team

/var/containers/Bundle/Application/{ApplicationUUID} [4], whereas the container that the application can read and manipulate in any way it wants is placed in /var/mobile/Containers/Data/{ApplicationUUID}. The following image below, from Apple's documentation archive, demonstrates this separation of containers:



Unlike macOS, containers on iOS are created when the application is installed rather than when opened [5]. After creating the container, the system makes sure to adjust the application's environment properly in accordance with the container (i.e., HOME and TMPDIR environment variables are adjusted to point to folders within the container).

The actual container directory is made up of 2 parts:

- A Property List (a format which stores structured data much like JSON, colloquially known as a "plist") file, which contains metadata describing the application & container. This Property List is created, managed and manipulated by containermanagerd, a daemon whose actual implementation resides in a private system framework by the name of ContainerManagerCommon.framework. The table below displays values that are documented by this Property List, and later retrieved by the system:

Key	Value
MCMMetadataIdentifier	Text describing the application's identifier
MCMMetadataInfo	A dictionary of general metadata information mostly pertaining to the Sandbox such as the Sandbox profile (SandboxProfileData) and

	other general information (SandboxProfileDataValidationInfo)
SandboxProfileData	Base64 Encoded Data of the application's Sandbox Profile, a concept to be explained later in this paper
SandboxProfileDataValidationInfo	A Dictionary containing information that will be used by the system (more specifically, libsandbox, more on this later in the paper) as inputs to compile the sandbox profile of the application (see above)

- The second part of the container directory is a folder called Data, where the application stores its data. The Data folder is structured in the same way that the user's home directory is (Documents, Library, Music, Desktop, Downloads, …). This is where the application will store its data.

The daemon mentioned above, containermanagerd, is extremely important when talking about sandboxing, as it communicates with both userland clients over one of Apple's IPC frameworks, named XPC, and with the kernel over a special Mach port (communication endpoints which could be used between both kernel and userland). The daemon acts somewhat as a bridge between the userland processes which install, restore, and delete apps, and between the kernel where Sandbox resides and needs information about an app's container (i.e., directory of the container), though its role is more than just a bridge, as its name describes, it also manages the actual containers and stores appropriate metadata in them.

Now that we have described the extremely basic layout of how Sandbox sets up its containerization, we will delve into *who* exactly gets sandboxed in the first place.

2. WHO GETS SANDBOXED?

The rule of thumb as to who is to be sandboxed differs by platform, and we will be looking at how it's done on macOS first. Before delving into exactly how the system determines that an application should be sandboxed or not, we will need to understand the concept of "Entitlements". Entitlements on Apple platforms simply describe the "capabilities" of an application (that the system should grant to that application, i.e., "this application needs print capabilities"), however, only applications signed by Apple can grant themselves any entitlements they need, while applications signed by anyone else can only use entitlements allowed by Apple.

In practice, entitlements are a Property List embedded into the binary of an application / tool which contains a dictionary of Key-Value pairs corresponding to each capability that the application should (or shouldn't) have, here's a simple example from a macOS system binary, TextEdit, the basic text editing app that comes pre-installed, much like Windows' Notepad (Dumped using the printents tool):

```
❯ printents /System/Applications/TextEdit.app
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>com.apple.application-identifier</key>
    <string>com.apple.TextEdit</string>
    <key>com.apple.developer.ubiquity-container-identifiers</key>
    <array>
        <string>com.apple.TextEdit</string>
    </array>
    <key>com.apple.private.hid.client.event-dispatch.internal</key>
    <true/>
    <key>com.apple.security.app-sandbox</key>
    <true/>
    <key>com.apple.security.files.user-selected.executable</key>
    <true/>
    <key>com.apple.security.files.user-selected.read-write</key>
    <true/>
    <key>com.apple.security.print</key>
    <true/>
</dict>
</plist>
```

What's relevant to us is one particular key, `com.apple.security.app-sandbox`, set to true, meaning that TextEdit voluntary requests the system to sandbox it (this is actually a bit unorthodox of entitlements, as they usually specify capabilities that the system should *grant*, rather than *restrict*, like the App Sandbox).

Upon the startup of any process on macOS / iOS, the dynamic linker (which loads all processes, known as "dyld") will load a system library by the name of libSystem.b, which initializes a couple of necessary System libraries, the important one in the case of sandboxing being libsystem_secinit. libsystem_secinit checks for if the process should be sandboxed (by checking for if the `com.apple.security.app-sandbox` entitlement is present and set to true on macOS, or if the `com.apple.private.security.no-sandbox` is present and set to false on iOS).

While the sandbox is voluntary for apps on macOS, where the developer can simply remove the `com.apple.security.app-sandbox` entitlement while packaging the app (except for apps distributed on the macOS App Store, which need to be sandboxed [6]), on iOS, however, the situation is different. Whether or not an application is sandboxed on iOS depends on where it was installed [7]. All iOS App Store applications (including system ones that can be deleted and re-installed, such as Maps), get installed to the location where they are sandboxed by default (/var/containers), thus App Store apps can't be un-sandboxed (unless an app possess and sets the `com.apple.private.security.no-sandbox` entitlement to false, however this is only limited to Apple's apps, meaning that 3rd party apps are always subjected to the Sandboxed on iOS with no exceptions). Certain system applications, such as Settings, are located outside of that directory (and are instead located in /Applications), and as such, the sandbox doesn't apply to those applications.

3. BEHIND-THE-SCENES WORKINGS

Before diving into how the Sandbox system is implemented, it's crucial to first dive in and understand how MACF (Mandatory Access Control Framework) works on iOS / macOS as well as how it communicates with its clients (one of which being Sandbox, however there are others as well). As mentioned in the start of this paper, Apple's MAC implementation is based off of TrustedBSD, introduced to macOS 10.5 [8]. The gist of how this MAC Framework works is that clients (in the case of Apple OSes, the clients are usually kernel extensions, i.e., Sandbox) register what is called a "policy". A policy contains callback functions (called "hooks") for events (called "operations") that the policy is interested in, hooks return either 0 to allow the operation to occur, or a non-zero value (almost always an errno code) in order to prevent the operation from occurring (i.e., a hook that would want to deny an operation to occur due to issues regarding permission would return EPERM). When the kernel executes an action that MAC policies could be interested in (i.e., allowing a userland process to write to a file), it iterates over all MAC policies that are interested in the event and checks for the return value of each policy's hook, *all* policies interested in an operation, with no exceptions, must allow the operation to occur, otherwise the kernel will prevent it from occurring. A policy signals to the kernel that it is interested in permitting or denying a certain type of operation simply by providing a hook for that operation.

Part of what makes MAC Policies incredibly powerful is just how many operations they are able to allow / disallow, as of the time of writing this (August 2024), there are 331 operations! With 39 being reserved. These operations include (but are not limited to)[2] control over:

- Filesystem related activities (Opening / Deleting / Moving files and directories, mounting, controlling whether a process can access a particular file / directory, etc.)
- Networking activities
- Accessing device drivers with IOKit (the framework which Apple provides as an interface to interact with hardware devices)
- Software HID (Human Interface Design) Control (i.e., being able to perform synthetic touches and keypresses)
- setuid, setgid, and other similar functions that processes can use to change their privileges
- A process' ability to debug another process or fork itself
- A process' ability to set Code Signing information

The flow of setting up a MAC policy is simple; a Kernel extension simply needs to create a structure consisting of information about the policy by the name of `mac_policy_conf`, clients must provide a display name (called `mpc_fullname`), a shorter name (called `mpc_name`), and a structure consisting of the hooks to operations that the client is interested in (`mac_policy_ops`), clients can also specify preferences for how the policy should be loaded (i.e., should this policy be loaded and initialized early in the boot process? Is it okay if the

[2] This list is, in no way shape or form, exhaustive. The full list can be found in the mac_policy.h file in the source code of the Darwin Kernel used on iOS / macOS, and the list is subject to change.

system unloads this policy?). After creating their policy configuration structure, clients simply need to call `mac_policy_register` and pass in the policy configuration as well as a pointer to a number which could later be used to de-activate the policy with `mac_policy_unregister`.

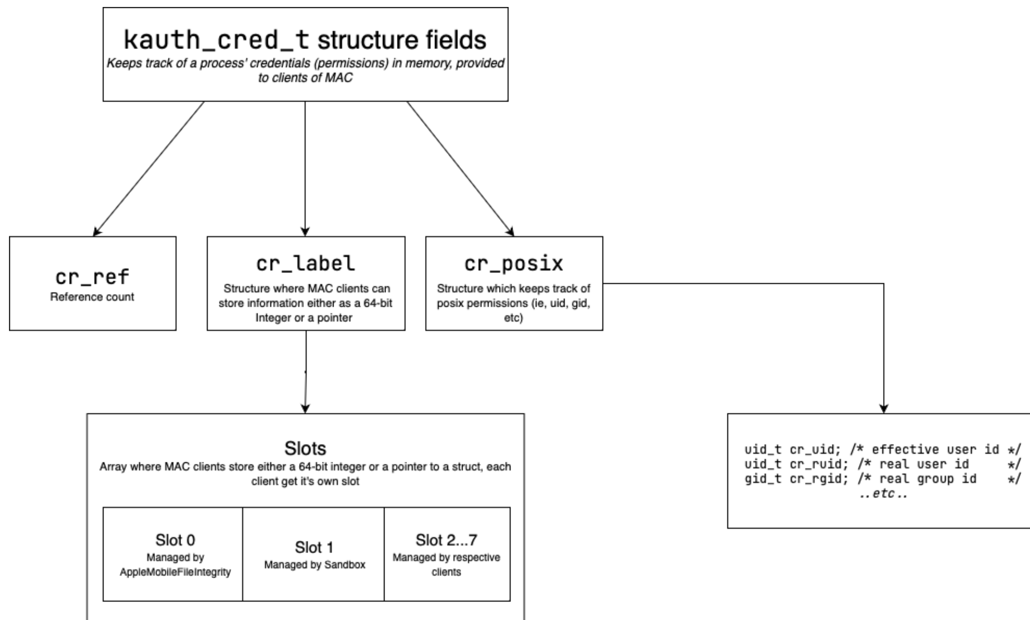
In a policy's hooks, information is provided about the process attempting the operation (process id, name, path, etc.) by the kernel, as well as credentials in the form of a structure to be discussed later (`kauth_cred_t`). Here is a simple demonstration of a policy, which prevents processes except ones named "OurSpecialProcess" from writing to a file if that file is named "PROTECTED_FILE":

```
#define PROTECTED_FILENAME "PROTECTED_FILE"
#define ALLOWED_PROCESS_TO_ACCESS "OurSpecialProcess"

/*
 this is a demo of MACF Policies,
 where we monitor for write events where the filename is PROTECTED_FILENAME,
 and if it is PROTECTED_FILENAME, then allow the write operation to occur only if the
process name is ALLOWED_PROCESS_TO_ACCESS
 */
static int policy_should_allow_write(kauth_cred_t active_cred, kauth_cred_t file_cred,
struct vnode *vp, struct label *label) {
    const char *vnodeName = vnode_getname(vp); /* get reference to name of file */

    /* check if it's our protected filename */
    if (vnodeName && strcmp(vnodeName, PROTECTED_FILENAME) == 0) {
        char procName[MAXCOMLEN];
        proc_selfname(procName, MAXCOMLEN); /* get name of process trying to write to
file */

         /* check if the process accessing PROTECTED_FILENAME is
ALLOWED_PROCESS_TO_ACCESS
         and if it isn't, return EPERM to signal that the process isn't allowed to
write to the file */
        if (strcmp(procName, ALLOWED_PROCESS_TO_ACCESS) != 0) {
            vnode_putname(vnodeName); /* release filename reference */
            return EPERM; /* deny operation to occur by returning EPERM */
        }
    }

    vnode_putname(vnodeName); /* release filename reference */
    return 0; /* return 0 to signal that we allow the operation to occur */
}

/* register that we want to monitor for and allow/deny write events */
static struct mac_policy_ops our_operations = {
    .mpo_vnode_check_write = policy_should_allow_write
};

static struct mac_policy_conf our_policy_configuration = {
    .mpc_fullname = "MACF Policy Demo",
    .mpc_name = "com.demo.macfpolicy",

    .mpc_ops = &our_operations
};

mac_policy_handle_t handle; /* handle to keep track of our MAC policy */
```

```
/* called when our kernel extension is started */
kern_return_t demo_kext_start(kmod_info_t *ki, void *data) {
    mac_policy_register(&our_policy_configuration, &handle, data);
    return KERN_SUCCESS;
}

/* called when our kernel extension is to be de-registered */
kern_return_t demo_kext_stop(kmod_info_t *ki, void *data) {
    mac_policy_unregister(handle);
    return KERN_SUCCESS;
}
```

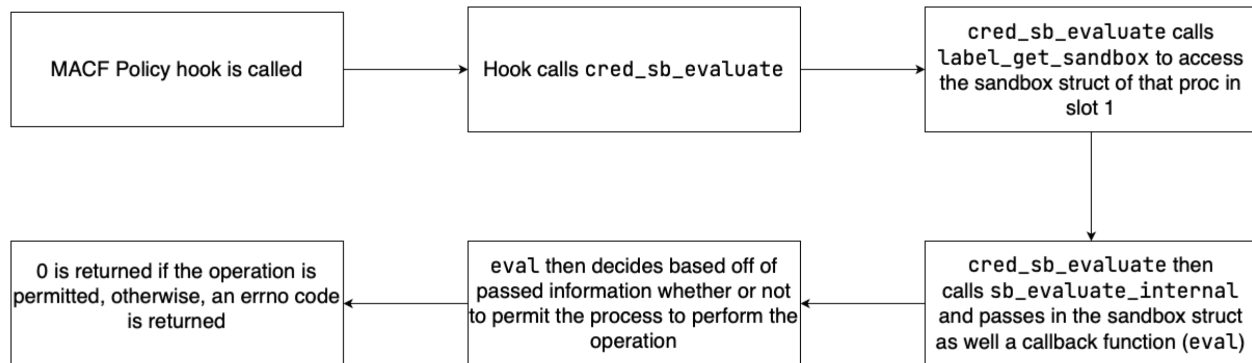
Kexts which provide policies also usually rely on userland daemons to do additional work and return status codes that signal whether or not it is appropriate to let the event proceed, in Sandbox's case, a userland daemon by the name of sandboxd exists on macOS (though not on iOS), mostly for the purposes of tracing [4], and it's role is very minor in comparison to other userland daemons with their kexts. The closest thing which Sandbox.kext has to a userland helper daemon which helps in an influential way is containermanagerd, as it is the daemon which deletes, creates, and manages containers as well as provide information about them to the kernel when requested.

The primary client of MAC policies is Sandbox (Sandbox also actually polices stuff in other ways than just MAC policies, Sandbox's operations don't necessarily refer to operations provided by MACF), which provides hooks for 159 operations (as of macOS 14.6.1). Just about every hook provided by Sandbox follows a similar flow of calling into one function, `cred_sb_evaluate`, which takes in 3 arguments: the first is always provided by MACF, and is a struct pointing to the credentials of the process attempting the operation (`kauth_cred_t`, who's anatomy is displayed below), the second argument is a number identifying the operation type (Operations policed by Sandbox), and the third is a buffer containing any additional information to pass to the function. `cred_sb_evaluate` calls onto `label_get_sandbox` to get a pointer to a struct where information about the process' sandbox is stored, the struct resides in slot 1 of the process' `kauth_cred_t` labels.



`cred_sb_evaluate` then calls onto another function with a pointer to the process' Sandbox structure, operation number, buffer, and a callback to `sb_evaluate_internal`.

`sb_evaluate_internal` then performs it's checks and balances with some help of the callback function passed into it, arguably being the star of the show, the `eval` function, a long, complex function which is around 1800 lines when decompiled. `sb_evaluate_internal` parses and checks some information, then calls and returns the return value of the callback `eval` function. All in all, the flow of a MACF hook from Sandbox is demonstrated below



The most important of these hooks is the hook for `cred_label_update_execve`, the largest hook by far coming in at 724 lines when decompiled. This hook is called when a process is spawned and is meant to signal to MAC clients to set the credentials of the process being spawned, in Sandbox's case, this is where it containerizes and sandboxes the spawned process, as well as where it creates the sandbox struct for the process to later store it in slot 1 of its `cr_label` [4]. It first starts by parsing specific entitlements from the process (a task delegated to one of Sandbox's friends, AppleMobileFileIntegrity) in order to determine the profile to use specified by the seatbelt-profiles entitlement if present (however this behaviour is exclusive to

iOS, as macOS instead uses Sandbox Profile Language files) and the ID of the application, then calls `platform_set_container` to set the container, creates the sandbox struct for this process, then creates what is called a Sandbox extension (to be explained furthermore in this paper) in order for the app to access it's own executable and container, calls `label_set_sandbox` to assign slot 1 of the process' cr_label slots to the sandbox struct in order to keep track of it in other hooks.

4. PROFILES

Due to the needs of Apple's sandboxed own apps (i.e., a sandboxed system app may need to communicate with a system daemon, which would be prohibited by the Sandbox under usual circumstances) and to provide flexibility, the Sandbox of an application can be configured with configuration files called profiles. A profile is made up of a set of rules where every rule consists of 3 core components: a *decision* to allow or deny an operation, the *operation(s)* to allow or deny, and optionally context to conditionally allow or deny the operation. Every Sandbox profile has a simple default allow/deny rule for operations which the profile doesn't cover. Here's an example of a profile:

```
;; Comments are declared like this
;; Begin example.sb
(import "system.sb")
(version 1)
(deny default)

(allow file-read* file-write* (home-subpath "/Documents/SecretDirectory"))
(allow process-exec (container-subpath "/helpertools"))
(deny device-microphone)
(deny device-camery)
;; End example.sb
```

The example Sandbox Profile above first has statements about the profile itself (importing another Sandbox profile, specifying the version, and the default statement), after that, we define our rules, first, we allow all file read and write operations to `~/Documents/SecretDirectory` (We don't know the actual home directory since this is just a configuration file, but we can use a macro provided by Sandbox, `home-subpath`), we allow the application to spawn binaries if they are under a folder named "helpertools" in the application's container, as well as unconditionally denying access to the device's camera and microphone.

In fact, the implementation of one of the most notorius security features of macOS, System Integrity Protection (collaquily known as "SIP" and "rootless"), designed to prevent malicious actors from modifying essential system files and directories (/System, /bin, /var, pre-installed system applications, etc) [9] is partially just a Sandbox profile called "platform_profile" [10] which resides in /System/Library/Sandbox/rootless.conf.

Sandbox profiles allow to specify arguments in a number of ways, including:

- String Literals (i.e., `allow file-write* (literal "/SomeFile")`), which will allow for all type of file writes for the path "/SomeFile"
- Variable Arguments (i.e., `allow file-write* (home-subpath "/SomeFile")`), which will allow for all type of file writes for the path ~/SomeFile under the home directory
- Regex Arguments (i.e., `allow file-write* (regex \.(doc|docx)`) which will allow for all type of file writes for the files with the extension "doc" and "docx"

Sandbox operations are divided into categories, for example, the file-write operations category consists of the following items:

- `file-write-acl` to permit or deny setting Access Control Lists
- `file-write-create` to permit or deny creating a specific file / directory
- `file-write-data` to permit or deny writing data to a specific file / directory
- `file-write-setugid` to permit or deny changing the owner uid or gid of a specific file / directory
- `file-write-unlink` to permit or deny deleting a specific file / directory
- …and some others not listed here for the sake of brevity.
-

Rules could either specify a specific operation from the category (i.e., if you'd like to allow / disallow just writing data, you could specify `file-write-data` in the rule), or could specify the category followed by * as an umbrella for all operations in the category (i.e., if you'd like to allow / disallow all forms of write operations, you could specify `file-write*`).

As previously mentioned, the operations which Sandbox polices aren't necessarily limited to the operations which MACF polices, in fact, Sandbox polices several more types of operations:

Non-MACF reliant Operation name (in Sandbox.kext)	What it polices
appleevent-send	Whether a process is allowed to send an Apple Event (a message-based form of IPC [11]) to other processes on macOS
lsopen	Whether a process is allowed to open another application directly using Apple's framework for managing applications, LaunchServices
nvram	Whether a process is allowed to set, get, and delete objects stored in the device's NVRAM (where the device stores persistent storage needed in updates, restores, and other use cases)
user-preference	Whether a process is allowed to get and set objects in a Preferences domain (a database used by applications to store user preferences) belonging to another application, for example, an application wanting to read the preferences of iMessage for whatever reason would add the following rule: `(allow user-preference-read (preference-domain "com.apple.Messages"))`

On macOS, Sandbox profiles are stored as plaintext files with the “sb” file extension on /System/Library/Sandbox/Profiles and are written in the human-readable format of the profile above, called the Sandbox Profile Language format, modeled after the Scheme language. On iOS, however, Sandbox profiles are stored in a binary format in the __TEXT.__const segment of Sandbox.kext’s binary, making them difficult to accurately dump and read, otherwise profiles work.

5. SANDBOX EXTENSIONS

Sandbox extensions are tokens (represented as a String) which a process can issue from the kernel representing an action (i.e. reading / writing to a specific path) that the process can then “consume” to perform that action, the process must be able to already perform the action that it is issuing a Sandbox extension for (i.e., if the process wants a Sandbox extension to read and write to /SomeFile, the process must already be able to read and write /SomeFile). On the surface, Sandbox extensions seem useless, why would a process need a token for an action it can already perform? However, they are incredibly powerful for their intended use case, where a privileged (often times completely un-sandboxed) process issues a Sandbox extension to a non-privileged, sandboxed process, which grants the sandboxed process the ability to perform that action which they otherwise wouldn’t have been able to.

The most apparent use case for this is the file selector panel on macOS, when an application opens the file selector panel, the panel itself is rendered and handled in a completely separate process, meaning that the process which requested the panel can’t read the files displayed on the panel (and therefore the sandbox still stays intact), when the user selects a file and presses “OK” to dismiss the panel, the file panel process (which is unsandboxed, and can therefore read whatever file chosen) issues an extension to the sandboxed process to allow it to read and write to the file selected, this is how, for example, Microsoft Word can read and write to documents that you open in ~/Downloads or in ~/Documents despite those files being outside of Word’s container.

Another use case of Sandbox extensions is a process which spawns un-sandboxed and wishes to later sandbox itself, however needs access to specific files / directories. In this case, the process can issue Sandbox extensions for those paths, “consume” the extensions, and later sandbox itself by calling `sandbox_init`.

Sandbox extensions are issued by calling two types of functions: `sandbox_extension_issue_(extension-type)` and `sandbox_extension_issue_(extension-type)_to_process`. The sole difference is that the former issues the extension for the current process, while the latter issues the extension for another process by passing in that process’ audit token (an audit token is a more secure way of identifying processes on Apple platforms, since process IDs are inheritly racy and vulnerable to re-use by another process).

When a process requests a Sandbox extension, the system passes in the type (as a number) and a string identifier called the “extension class” which narrows down the specific ability the client

wants to issue (i.e., the extension class for files specifies if they want just read-only or read / write capabilities), Sandbox extension types include:

ID	Sandbox extension Type	Purpose
0	file	read / write access to a specified file, read only access can be issued by passing in com.apple.app-sandbox.read as the extension class, and both read / write access can be granted by passing in com.apple.app-sandbox.read-write as the extension class
1	mach	Access to specified Mach ports / XPC services
2	iokit_user_client_class	Access to specified IOKit User Client class
3	generic	Unknown (although it's name indicates that it *could* be an umbrella for other operations policed by Sandbox?)
4	posix	"Access to named POSIX IPC object (UN*X sockets, etc)" [4]
5	preference	(Supposedly) Read / write access to a specific preference domain (see NSUserDefaults), however couldn't get this to work in practice
6	sysctl	(Supposedly) Read / write access to a specific sysctl values, however couldn't get this to work in practice

Sandbox extensions are not automatically in effect once they are issued, but rather, they must be activated (the term Sandbox uses for this is "consume") by calling the `sandbox_extension_consume` function and passing in the token String that was issued by the kernel, the consume function returns a 64-bit integer as a handle identifying the extension, which the process can then use to relinquish the effect of the Sandbox extension by calling `sandbox_extension_release` and passing in the handle.

It is important to note, however, that while a process can *issue* a Sandbox extension for another process, the target process has to be the one that activates it, this is usually done by the sending the the extension string to the target process over IPC, where the target process will then activate the extension itself, such as in the case of the file panel on macOS, which sends a sandbox extension string to the client application when a file / directory is chosen, and the application itself activates the extension (however this is handled automatically by the AppKit framework in the case of the file selector panel, which means developers don't usually have to worry about these details).

The code below demonstrates an example of a process issuing a Sandbox extension for another process, activating the extension by consuming it, running a callback function to signal to the target process to perform tasks to the path requested, then after the callback function is done, the Sandbox extension is released in order to relinquish it's effects.

```
bool grant_sandbox_extension_to_path_for_process(const char *path,
                                                 audit_token_t target_process_audit_token,
                                                 void (^callbackWhileExtensionIsInEffect)(void))
{

char *extToken = sandbox_extension_issue_file_to_process("com.apple.app-sandbox.read-write",
path, 0 /* additional flags, anything other than 0 returns "invalid" */,
target_process_audit_token);
```

```
    if (extToken == NULL || strcmp(extToken, "invalid") == 0) {
        // we couldn't get a valid Sandbox extension, return false
        return false;
    }

    int64_t extensionHandle = sandbox_extension_consume(extToken);
    if (extensionHandle == -1) {
        // sandbox_extension_consume returns -1 when an error occurs,
        // and sets the errno value to indicate why the error occured
        printf("grant_sandbox_extension_to_path_for_process: failed to consume extension, reason: 
%s\n", strerror(errno));
        return false;
    }

    // now that the extension is in effect,
    // let's call our callback function that signals to the process
    // to execute code while the extension is in effect
    callbackWhileExtensionIsInEffect();

    // Release the extension
    sandbox_extension_release(extensionHandle);

    return true;
}
```

The first part of the Sandbox extension string (up until the first semicolon) is a hash of 2 things: the rest of the string as well as an 128-bit secret which the device generates at boot (meaning that Sandbox extensions cannot be stored and re-used after reboot), and the second half of the string is a description of the Sandbox extension itself, including the type of extension, the extension class, information identifying the target process, and additional data (i.e., for file Sandbox extensions, the additional data would be the path to the file to get read or read / write abilities to). [12]

The image below demonstrates the segments and anatomy of a Sandbox extension string:

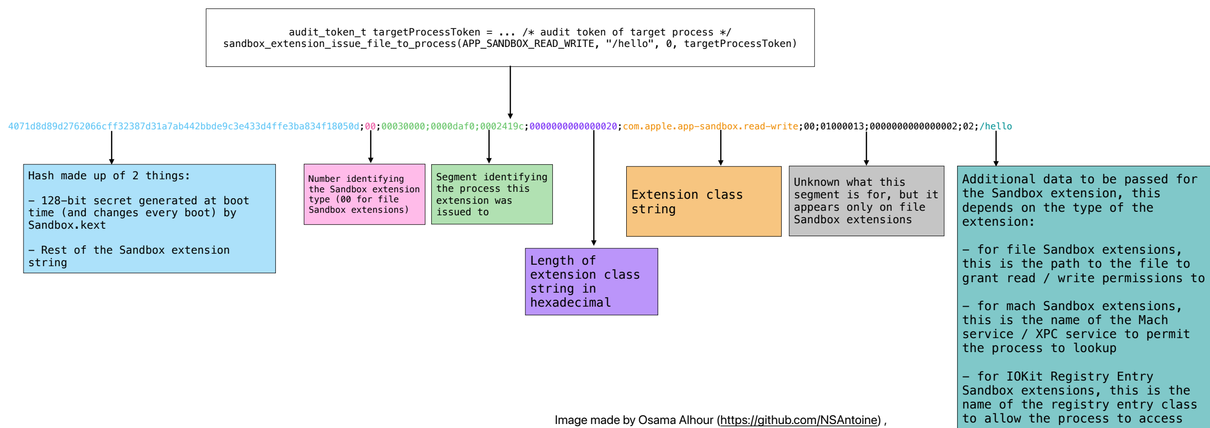


Image made by Osama Alhour (https://github.com/NSAntoine) , "A Worm's Look Inside: Apple's Sandboxing security measures on macOS & iOS"

The `sandbox_extension_issue_(extension-type)`, `sandbox_extension_consume`, and `sandbox_extension_release` functions all follow a very similar flow of communicating to the Sandbox kext from userland, a concept which will be discussed in the following section.

6. USERLAND INTERACTION WITH SANDBOX.KEXT

There are various APIs (though most are private) which allow for some sort of interaction between the userland process and Sandbox.kext, these APIs are implemented in a system library known as libsandbox. These include, but are not limited to:

- `sandbox_init / sandbox_init_with_parameters` to initialize the sandbox of a process
- `sandbox_extension_issue_(extension-type)` and `sandbox_extension_issue_(extension-type)_to_process` family functions to issue Sandbox extensions
- `sandbox_extension_consume` to activate an issued Sandbox extension
- `sandbox_extension_release` to relinquish an activated Sandbox extension
- `sandbox_inspect_pid` to dump information Sandbox information about a specific process (though it seems this function has been removed in newer versions of iOS / macOS)

The question, then, is how do these functions, from userland processes, communicate with Sandbox.kext, a Kernel extension? MAC policies can provide an interface for userland processes to interact with them, on the policy's side, this simply involves registering for the `mpo_policy_syscall` operation in the policy's configuration, where clients are provided with a number identifying the action that the userland process is requesting the MAC policy client (in this case, Sandbox.kext) to perform, as well as a pointer to additional data provided by the userland process. As the operation's name implies, this works similarly to how actual kernel syscalls work. On the userland process' side, the process calls a function by the name of `__mac_syscall`, and passes in 3 arguments:

- The name of the MAC policy to send the syscall to (in Sandbox's case, this is always just going to be "Sandbox")
- The syscall number (policy-specific)
- A pointer to additional data that the MAC policy may need (policy-specific)

When called, `__mac_syscall` performs an *actual* syscall to the *kernel itself*, the kernel then iterates over all registered MAC policies to find one where the policy's name (as the policy defined in the `mpc_name` field of it's `mac_policy_conf`) matches the name passed in by the userland process, then calls the policy's `mpo_policy_syscall` function, passing in the syscall number and the pointer to the additional data provided, as well as a struct pointing to the process requesting the syscall (if the policy names matches but it does not implement `mpo_policy_syscall`, or no policy with the specified name was found, errno 103 (`ENOPOLICY`) is returned).

In the case of the libsandbox functions listed above, they call into a `__sandbox_ms` function, which in implementation is the exact same as `__mac_syscall`:

```
// actual implementation
int __sandbox_ms(const char *policyName, int syscall_number, void *arg) {
    return __mac_syscall(policyName, syscall_number, arg);
}
```

A simple function to demonstrate this is `sandbox_extension_consume`, the function which requests Sandbox.kext to activate a specified Sandbox extension and returns a handle identifying the activated extension as a 64-bit integer. Decompiling the function (and cleaning up the output) results in this:

```
int64_t sandbox_extension_consume(const char *ext) {
    int64_t handle;

    int64_t args[] = {
        (int64_t)ext, /* memory address of extension token string */

        (strlen(ext) + 1), /* size of the extension token string + 1 to
include NULL terminator character */

        (int64_t)&handle /* memory address of the handle for Sandbox.kext to
write to */
    };

    int ret = __sandbox_ms("Sandbox", 6 /* syscall number for
sandbox_extension_consume */, args);
    if (ret == 0) {
        return handle; // syscall succeeded, return handle
    } else {
        return -1; // didn't succeed, return -1 :(
    }
}
```

In this example, the function creates an array of arguments to pass into the Sandbox syscall, consisting of the memory address of Sandbox extension string, the size of the extension string (required in order for Sandbox.kext to copy the string to kernel memory from user-space memory to read it), and the memory address where we want the handle number to be written to. When the kernel receives a `__mac_syscall` syscall for Sandbox, it'll iterate over all registered policies, find Sandbox's policy, and call it's `mpo_policy_syscall` function, Sandbox.kext will then call the appropriate function for the syscall number provided (in the case of `sandbox_extension_consume`, whose syscall number 6, the function is `syscall_extension_consume`).

7. CONCLUSION

In this paper, we've explored the interface and model of Apple's notoriously robust sandboxing system on iOS and macOS, as well as internal implementations from the side of the kernel, userland, and Sandbox.kext itself, how Sandbox.kext interacts with the rest of the system, and how Apple configures the Sandbox differently based on an application's needs with profiles.

Works Cited

[1] National Institute of Standards and Technology, mandatory access control (MAC) - Glossary: CSRC. https://csrc.nist.gov/glossary/term/mandatory_access_control

[2] Red Hat, Inc 4.2. SELinux and Mandatory Access Control (MAC). https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/virtualization_security_guide/sect-virtualization_security_guide-svirt-mac#sect-Virtualization_Security_Guide-sVirt-MAC

[3] FreeBSD, TrustedBSD – Home http://www.trustedbsd.org

[4] J. Levin, Hack in the (sand)Box https://newosxbook.com/files/HITSB.pdf

[5] Apple, Inc, File System Basics. https://developer.apple.com/library/archive/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html

[6] Apple, Inc, App Sandbox | Apple Developer Documentation. https://developer.apple.com/documentation/security/app_sandbox/#

[7] J. Levin, The Apple Sandbox: Deeper Into The Quagmire - Jonathan Levin (Video presentation at Hack In The Box Security Conference). https://youtu.be/mG715HcDgO8?si=fFAbfbWshpqKqg5W&t=1092

[8] A. Stavonin, Working with TrustedBSD in Mac OS X | System Development. https://sysdev.me/trusted-bsd-in-osx/

[9] Apple, Inc, About System Integrity Protection on your Mac - Apple Support. https://support.apple.com/en-us/102149

[10] HackTricks, macOS Sandbox | HackTricks. https://book.hacktricks.xyz/macos-hardening/macos-security-and-privilege-escalation/macos-security-protections/macos-sandbox

[11] SwiftAutomation, Understanding Apple events. https://hhas.bitbucket.io/understanding-apple-events.html

[12] opa334, Sandbox Extensions explained. https://github.com/opa334/sandbox_extension_generator?tab=readme-ov-file#sandbox-extensions-explained