Credentials Transitions with `mdo(1)` and `mac_do(4)`

BY OLIVIER CERTNER

In this article, we explore how the **`mdo(1)`** program can be used to easily and quickly launch a new process with different credentials and how system administrators can enable credentials transitions initiated by unprivileged users by leveraging the **`mac_do(4)`** kernel module, obviating the need to install third-party programs such as **`sudo(8)`** or **`doas(1)`** in simple role-based scenarios.

The traditional UNIX approach to access control essentially relies on the following concepts and components:

- Users and groups. Groups are meant to ease administration by treating all users of a group uniformly in some ways.
- Processes, as subjects acting on behalf of some user and groups, which are referred to as their credentials.
- File ownership (one user, one group) and permissions, which separately guard accesses by the owner, by members of the file's group, and by other users.
- The special **`root`** user[1], which has all privileges and in particular is not subject to access control.
- Set-user-ID/set-group-ID executables, which, when launched, have their process respectively endorse the executable's owner as the user and the executable's group as the "primary" group[2].

One of the primary duties of a system administrator is to give their users appropriate access to various resources of the systems. This, for the most part, translates into defining users and groups and ensuring that files have the proper permissions with respect to the expected security policy.

The UNIX access control model has the flexibility that users need not represent real, human individuals, but may as well represent roles that can be assumed by multiple real users who require access to specific resources and information. Such a role-based approach relying on UNIX users instead of just groups is, in fact, necessary in all but the simplest file-sharing scenarios. It makes temporarily adopting another set of credentials, established from a target user, an important feature of the system, which the **`su(1)`** program traditionally fulfills.

One of the primary duties of a system administrator is to give their users appropriate access to various resources of the systems.

However, **`su(1)`** requires authenticating to a new user before switching to its credentials, typically by asking for the user's password[3], which is not convenient either for humans who have been assigned roles and are already authenticated or for automated scenarios. It nec-

essarily spawns the target user's shell, which precludes using it with users who have no valid login shells, whereas this is typically desired for role users that nobody should be able to log in as directly. It also makes launching a specific program with arguments more cumbersome than it should be[4].

In order to overcome these limitations, system administrators commonly install other programs designed to run commands on behalf of other users such as **sudo(8)** or **doas(1)**. However, programs like **sudo(8)** have a non-negligible attack surface, in part due to their number of infrequently used features, and, in particular, modularity, which can be dangerous from a security standpoint. More generally, having programs installed with the executable file's owner being **root** and having the set-user-ID mode bit set is a security concern, as compromising them can mean gaining full administration rights through code execution as the **root** user. But traditional UNIX did not provide any other way to change credentials, which is why programs like **su(1)** and **login(1)** are installed this way.

As an alternative to executables with the set-user-ID mode bit set, more colloquially called "setuid executables", we provide the **mac_do(4)** kernel module, built on top of FreeBSD's MAC framework[5]. Its purpose is to allow only certain credentials transitions from unprivileged processes, thus not requiring the corresponding executable image to be installed "setuid".

As an alternative to "setuid executables" in role-based settings, we provide the **mac_do(4)** kernel module and its **mdo(1)** companion program.

mdo(1) is the companion program to **mac_do(4)** and actually requests the desired credentials transitions to the kernel. **mdo(1)** can be used standalone by the **root** user who has all privileges. Otherwise, its requests are vetted by the **mac_do(4)** kernel module according to the administrator's configuration.

In this article, we first describe how to use **mdo(1)** to launch commands under new credentials, walking through a series of examples. Then, we explain how to configure **mac_do(4)** to enable specific credentials transition in support of role-based schemes, on the host system as well as in jails, also offering some insights on the current design. Finally, we ask for user feedback on what should come in the relative short term and possible longer term future plans.

Using mdo(1)

mdo(1) is designed to run any command with an arbitrary set of credentials. If you have not yet configured **mac_do(4)**, which is covered in the next section, you can still run all the examples below as **root**. For most examples, FreeBSD 15.0 is required as FreeBSD 14.3's **mdo(1)** only supports options **-u** and **-i**[6].

For safety reasons, the target process credentials have to be fully specified, either explicitly by listing all users and groups and their requested values, or indirectly by establishing a baseline that provides a default value for each of them and can then be amended by additional options.

In a role-based setting, the most common use case is arguably to endorse the credentials of some user as if he has just logged in. So, **mdo(1)** supports it in the simplest form

possible, with the only needed option being **-u** (for "user") to establish a baseline from the named user, as in:

```
$ mdo -u www /usr/local/bin/occ
```

To **sudo(8)** and **doas(1)** users, this command-line should strike you as extremely similar to what you use with those tools: Basically, **mdo** replaces **sudo** or **doas**, and the rest is identical.

Obviously, this cannot work if passing some user numerical ID as opposed to a user name, as the full login credentials are determined by the password and group databases, which are indexed by names[7]. Using **-u** with a user numerical ID only specifies the user (actually, the real, effective and saved user IDs), and **mdo(1)** then needs to be told about all target groups. That has to be done either explicitly as we will see below, or through **-i** which means to use the current groups as a baseline, else **mdo(1)** will simply error.

Keeping the current groups, such as in:

```
$ mdo -u 10002 -i
```

can be useful, e.g., if you need to temporarily process alien files whose owner is not in your password database.

-i also works with **-u** with a user name, and it means the same. E.g., if **foo** is some user in your password database that has ID 10002, then this command is entirely equivalent to the previous example:

```
$ mdo -u foo -i
```

Suppose now you want to explicitly specify groups, either because you are using a numerical user ID as we saw above, or because you want to override the groups associated with some user. You can use:

- **-g**: To set/override the primary groups[8].
- **-G**: To set/override the full set of supplementary groups. The comma-separated list you provide here is considered complete, i.e., it should contain all supplementary groups. Keep in mind that, starting with FreeBSD 15, a user logging in has their initial group, as specified in the password database, also in its processes' supplementary groups set.
- **-s**: To amend the supplementary groups set. The argument for this option consists of a list of comma-separated directives. You can ensure a group is part of the supplementary groups with a **+** directive, or ensure it is not with a **-** directive, or reset the list with a **@**, making **-s** work like **-G** but with a different syntax[9].

Some examples:

```
$ mdo -u unprivileged_user -g wheel -G wheel,staff,operator
```

starts a shell as user **unprivileged_user**, but ignores the groups specified in the password and group databases, replacing them with the passed ones. **-g** and **-G** are useful for testing which rights would be given to a user with a particular set of groups.

However, if the point is only to log in as some user with some additional groups, e.g., **wheel** and **operator**, in an "augmented" role scenario, **-s** is the option to use:

```
$ mdo -u unprivileged_user -s +wheel
```

or, conversely, if membership of some group must be temporarily revoked, e.g., when this group is used as a tag to deny access through **ugidfw(8)** (and **mac_bsdextended(4)**) for access control:

```
$ mdo -u unprivileged_user -s -tag_group
```

If the user should not change, there is no need to specify it explicitly with **-u**. Instead, you can just use **-k** (for "keep"), meaning to start with all current users and groups as the baseline. **-k** is exclusive with **-u** and implies **-i** (start from current groups).

Note that, in all cases, it is possible to override the credentials' groups using the explicit options we have seen (**-g**, **-G**, and **-s**).

Finally, the real, effective and saved variants of users and primary groups can be separately overridden if needed, using **--ruid**, **--euid** and **--svuid**, and **--rgid**, **--egid** and **--svgid** respectively. When all three variants are specified, there is no need to respectively use **-u** or **-g**, although specifying **-u** with a name is still useful to get its associated groups.

As you can see, in addition to its simplicity for the most common use cases, **mdo(1)** thus has the advantage over **su(1)**, **sudo(8)** or **doas(1)** that it allows to control every aspect of the target credentials, making it the tool of choice to test or temporarily use arbitrary credentials in advance of a modification of the password and group databases, or for role-based settings where endorsing a role comes from being part of additional groups rather than switching users.

mdo(1) is only concerned with changing credentials. Consequently, its code is relatively simple, and a special effort was made to make it as clear and as minimal as possible while being "obviously" correct. This makes for a program that is easy to audit and results in a very small binary, weighing a little more than 7kB on my **stable/14** machines. Compare this to **doas(1)** which weighs a little more than 27kB, and **sudo(8)** at 229kB completed by **sudoers(5)**, its default security policy plugin, at 628kB, both programs being installed as "setuid executables" in contrast to **mdo(1)**.

Currently, as it is geared to role-based scenarios, **mdo(1)** does not ask for any password or other form of authentication when requesting new credentials, instead relying solely on the requester's credentials for this purpose. As one of the possible future directions, listed in the conclusion of this article, we may add support for asking the current logged-in user's password. Additional functionalities related to switching to another user (such as login classes, login name, scheduling priorities, etc.) may also be considered depending on user feedback.

Configuring mac_do(4)

For a non-root user to be able to leverage **mdo(1)**, configuring **mac_do(4)** is compulsory since **mdo(1)** is by-design not installed "setuid".

mac_do(4) is not compiled in the kernel by default, but can easily be loaded as a module:

```
# kldload mac_do
```

You can then access its parameters below the **security.mac.do sysctl(8)** node. Currently (FreeBSD 14.3 and 15.0), the following are available:

- **enabled**: Whether the module is enabled (defaults to true). This is a global toggle. It is possible to deactivate **mac_do(4)** selectively on the host system or in any jail via rules (next knob) or jail parameters (see corresponding subsection below).
- **rules**: A list of rules indicating which credentials transitions are allowed. We are going to study several examples in the next subsection. **rules** has an empty value by default, meaning that **mac_do(4)** will not allow any credentials changes by itself.
- **print_parse_error**: Whether to print a parse error on the console and system log when setting rules fails.

Let's start by illustrating rules, and we will then get to how to configure jails.

Rules

Related to some examples we gave above for **mdo**, let's authorize user **unprivileged_user** with UID 10001 to endorse the **www** user (UID 80) representing a webmaster role:

```
# sysctl security.mac.do.rules='uid=10001>uid=80,gid=80,+gid=80'
```

In this example, there is only a single rule. The **>** token separates both parts of a rule, the left part being the "from" one, also called "match", and the right part being the "to" one, also called "target". **:** has been the historical separator token and still works, but we felt that **>** makes for more easily readable rules, especially to UNIX-trained eyes that can easily interpret **:** as a list separator between similar elements. Because **>** is a shell special character, you need to quote it somehow. For simplicity, we advise to always quote the value passed to **sysctl(8)**. Any amount of spaces can be used between tokens as another slight help to human users, and this feature also requires shell quoting.

The "from" part (**uid=10001** in the above rule) is pretty straightforward and indicates to match processes whose user ID[10] is 10001, thus matching **unprivileged_user** (and possibly other users with the same user ID). Note that only numerical IDs are allowed, not user names. The kernel indeed does not know about user names, which are irrelevant credentials-wise.

The "to" part (**uid=80,gid=80,+gid=80**) is a bit more involved. It contains three clauses separated by **,**. The **uid=80** and **gid=80** ones should be pretty straightforward: They allow switching to **www** in terms of user and initial ("primary") group ID. The last clause, **+gid=80**, is about supplementary groups, and says that 80 as a supplementary group ID is allowed but not mandatory. In general, **gid** preceded by a flag, here **+**, applies to supplementary groups. Other possible flags are **!** and **-**, and will be illustrated below.

Such a rule allows, e.g., the example command we saw in the previous section to be executed by **unprivileged_user**:

```
$ mdo -u www /usr/local/bin/occ
```

Note that the **uid=10001>uid=80,gid=80,+gid=80** rule is quite stringent, and for example would not allow **mdo -u www** to succeed if, e.g., user **www** was also a member of another group than **www**, as **mdo -u www** would try to install the supplementary groups mandated by the password[11] and group databases, and that other group does not appear in the rule.

It also forbids, e.g., **mdo -u www -i**, meaning to switch to user **www** but to keep the current groups, presumably those associated to **unprivileged_user** if they were not changed in the meantime. If an administrator wants this to work, it needs to relax the checks on groups. Assuming **unprivileged_user** is only a member of a group with the same name and GID 10001, they could use:

```
# sysctl security.mac.do.rules='uid=10001>uid=80,gid=80,gid=10001,+gid=80,+gid=10001'
```

As you can probably infer from this example, specifying multiple target clauses with **gid** and **+gid** means that any of the specified groups can be present in the target credentials[12].

In addition to the two last **mdo(1)** use cases, this last rule also allows **unprivileged_user** to become **www** while endorsing both groups 80 and 10001 at the same time[29]. If this is not desired at all, then the following setting could be used instead:

```
# sysctl security.mac.do.rules='uid=10001>uid=80,gid=80,+gid=80;uid=10001>uid=80,gid=1
0001,+gid=10001'
```

This time, there are two rules separated by **;**. When there are multiple rules, it is enough for one of them to validate the transition for it to be possible[13]. This setting still allows for mdo **-u www** and **mdo -u www -i** to work while ruling out something like **mdo -u www -i -s +www** or **mdo -u www -g 10001**.

If for some reason **mdo -u www -i** should work also when the current groups do not reflect what the databases say for user **unprivileged_user**, you can alternatively use:

```
# sysctl security.mac.do.rules='uid=10001>uid=80,gid=80,+gid=80;uid=10001>uid=80'
```

The second rule above, **uid=10001>uid=80**, allows a change of user ID without changing the current groups, so is exactly adapted to the use of **-i** with **mdo(1)** when any set of current groups can be kept. That second rule is in fact a shortcut for **uid=10001>uid=80,gid=.,!gid=.**, where **.** stands for the current primary groups[2] in the case of **gid**, and for the current supplementary groups in the case of **!gid**, and more broadly in the case of **gid** preceded by another flag. Note that this default part, **gid=.,!gid=.**, is implied only when no target clause has **gid** as its type (with or without flags). In particular, the following rule: **uid=10001>uid=80,gid=.** would prevent any switch that does not drop all supplementary groups, as no **gid** clause with flags appear.

An additional **gid** flag, **-**, can be used to indicate that a group shall not be part of the final supplementary groups. You may at first find this strange, as allowed groups have to be explicit in rules, barring the default explained in the previous paragraph. This is actually useful in conjunction with **.** used with **+gid** or **!gid**, in order to rule out some of the current groups. For example, if you want to allow **unprivileged_user** to switch to user **www** but retaining its current groups while ensuring that **wheel** does not appear in the final supplementary groups, instead of the above **uid=10001>uid=80**, you could use **uid=10001>uid=80,gid=.,+gid=.,-gid=0**[14].

Finally, in place of user or group IDs in rules, you can use ***** or **any** to mean any possible ID. For example, if you do want to allow members of **wheel** to become **root**, you could use a rule like **gid=0>uid=0,gid=*,+gid=***, basically saying that any set of target groups will do. Going even further, if you do not want to impose switching to **root** before becoming another user, you could as well use **gid=0>uid=*,gid=*,+gid=***, which can be abbreviated to **gid=0>any**. Let us remind you that, currently, **mdo(1)** is geared to role-based schemes and consequently, as in any other case, will not ask for a password to be entered to switch to another user, even if the latter is **root**.

When there are multiple rules, it is enough for one of them to validate the transition for it to be possible.

We have just demonstrated a large practical assortment of possibilities offered by **mac_do(4)**'s rules, which as you can see are very flexible and able to express precisely the target credentials that are to be allowed[15]. We have tried hard to keep the syntax as easy as possible to understand with the constraints of an essentially single-line **sysctl(8)** value, imply-

ing terseness, sufficient expressive power and the kernel dealing only with numerical IDs and not accessing the password and group databases. Even if you do not immediately grasp what a particular setting of **`security.mac.do.rules`** means, it should not take long before you do, so do not get overwhelmed by the examples and take some time to study them as necessary.

An exhaustive and more formal specification of rules can be found in the **`mac_do(4)`** manual page.

Jails

Jails in FreeBSD form a hierarchy[16], whose top is the host system[17]. Each individual jail has parameters, some of which can only be set at jail creation, and others also while the jail runs, from outside the jail.

`mac_do(4)` supports per-jail configuration thanks to the following parameters:

- **`mac.do`**: The per-jail module's mode.
- **`mac.do.rules`**: The per-jail rules that apply.

Parameter **`mac.do.rules`** contains the applicable rules, with exactly the same format as the **`security.mac.do.rules sysctl(8)`** knob we saw in the previous section.

It is usually desirable to control security parameters from outside a jail, and that is actually the only possibility at jail creation. However, it is also useful to have jails behave as closely as possible to the host system. Since **`mac_do(4)`** is a tool for an administrator to authorize credentials transitions, an administrator in a jail should also be able to use it.

For this reason, the **`security.mac.do.rules sysctl(8)`** knob was made jail-aware, i.e., it reflects the current jail's setting and can be set from the jail itself. **`security.mac.do.rules`** inside a jail and the corresponding jail's **`mac.do.rules`** parameter are in fact the same variable, so their values are always the same. An outside modification of **`mac.do.rules`** is immediately in force inside the jail, and conversely reading the jail parameter reveals any inner modification to **`security.mac.do.rules`**.

Parameter **`mac.do`** indicates how **`mac_do(4)`** works in a jail. As typical for the master parameter of a module supporting jails, it accepts or reports the following values:

- **`new`**: Jail's configuration is independent from that of the parent jail.
- **`inherit`**: Jail's configuration is inherited from the parent jail.
- **`disable`**: **`mac_do(4)`** is disabled in the jail.

For obvious security reasons, the default value is **`disable`**, except if **`mac.do.rules`** is explicitly set.

You may wonder what the exact interactions of this parameter with **`mac.do.rules`** are, as both parameters appear to be somewhat redundant. As said in this section's introduction, setting rules to an empty string causes **`mac_do(4)`** to ignore credentials change requests, and since rules are per-jail, this also works as a per-jail toggle to disable **`mac_do(4)`**, similarly to the **`disable`** value. Conversely, setting **`mac.do.rules`** from outside the jail, or **`security.mac.do.rules`** inside it, always has the effect of establishing per-jail settings, which conceptually corresponds to **`new`**.

We introduced[18] the **`mac.do jail`** parameter for two reasons. First, most kernel modules supporting jails provide a single knob to enable or disable its functionality inside a jail, and we found it good to have one, both for system consistency but also to provide a perhaps more natural way of disabling **`mac_do(4)`** than setting the rules to an empty string. Second, it

gives us the opportunity to introduce a new inheritance mode, through the **inherit** value, which can be very useful to administrators who want a set of jails to behave the same.

Before examining what inheritance exactly means, let's first see how **mac.do** and **mac.do.rules** stay consistent. Internally, each jail holds a kind of flag indicating whether it inherits from its parent and, if it does not, a copy of the rules setting (**mac.do.rules**) and an internal representation for them, avoiding any information redundancy[19]. We do not actually store any value that directly corresponds to the **mac.do** parameter. Rather, the latter gets synthesized from the available data when it is read. After this description, you probably have an idea of how it goes: If the inheritance flag is set, then reading **mac.do** returns **inherit**, else, if no rules were specified (empty string), **disable**, else **new**. When setting **mac.do** explicitly, **mac_do(4)** checks that its value is consistent with that of **mac.do.rules**. If **mac.do** is set to **new**, **mac.do.rules** must be specified. For the other cases, we apply the robustness principle[20], tolerating the presence of **mac.do.rules** with an empty string in jail parameters, even if strictly speaking it should be absent.

We introduce a new inheritance mode, which can be very useful to administrators who want a set of jails to behave the same.

When setting **mac.do** to **inherit**, **mac_do(4)** simply uses the configuration that applies to the parent jail, which itself may come from a jail higher in the tree. The main consequence is that a change of rules in any of the parent jails up to the first that does not inherit automatically and immediately does apply in a jail with **inherit**. This alleviates the administrator from having to change the configuration of multiple jails in a tree when all of them are supposed to stay in sync. As already noted, explicitly setting rules on a jail, whether through **mac.do.rules** or **security.mac.do.rules**, establishes independent per-jail settings, effectively breaking inheritance. Re-enabling it later is always possible, by just setting **mac.do** to **inherit** again.

As with any jail parameters, you can use these to easily configure a jail at its creation, either directly on **jail(8)**'s command-line, e.g.:

```
jail -c name=test_jail path=/ mac.do=inherit
```

or through **jail.conf(5)**. To modify some parameters as the jail is running, use **jail -m** as usual, e.g.:

```
jail -m name=test_jail mac.do=disable
```

Boot-up

Since **mac_do(4)** configuration on the host happens via **sysctl(8)** knobs that are also tunables, you can use either of the two different mechanisms that the base system provides to set them at boot.

As a first possibility, you can tune your **loader.conf(5)** configuration, by adding a line like:

```
security.mac.do.rules='uid=10001>uid=80,gid=80,+gid=80;uid=10001>uid=80'
```

This is adapted to cases where rules really have to be available very early at boot, or, e.g., if you are not using the base system's **rc(8)** bring-up framework.

Else, you can add the exact same line to `sysctl.conf(5)`, and the `sysctl(8)` knob will be set accordingly when `rc(8)` executes[21].

We have had some limited feedback that a few people do not find it very practical that `mac_do(4)` only deals with numerical IDs and find the `sysctl(8)` knob syntax quite terse. These are essentially the consequences of having the transition rules in the kernel, which allows to cope with a strong threat model where some userland parts may have been compromised. However, we understand that most people do not require such a level of security, and that having userland tools produce final rules from references to the content of the password and group databases could be useful to them. There has been a proposal in this direction consisting of a dedicated executable and configuration file for `mac_do(8)` which for the moment is stalled as, among others, we have been reflecting on the overall design, including how to organize executables and possible future configuration files and how to avoid conflicts. If more people are interested in such functionality, we may make progress on this front sooner rather than later. See also, in this article's last section, the short-term features we plan to add to `mdo(1)`, one of which will bridge an important part of the gap.

Some Notes on `mac_do(4)`'s Design

Baptiste Daroussin initially launched the `mac_do(4)`/`mdo(1)` project with the goal to enable role-based credentials transitions without using "setuid executables". In high-security, norm-constrained settings, installing these executables, if at all possible, may be subject to long and complex security audits, which often need to be renewed as the executables are upgraded. Thus, `mac_do(4)` was conceived as a kernel-based alternative that, thanks to the MAC framework[5], can authorize unprivileged processes to successfully change credentials. In addition to alleviating the need for "setuid executables", this architecture instantly reduces the impacts of a successful attack on or a programming bug of credentials-changing programs.

The original implementation of `mac_do(4)` only monitored the `setuid()` system call, authorizing a specific call to it according to rules matching the original user and the target one. In order to allow `mdo(1)` to change groups as prescribed by the password and group database for the target user, `mac_do(4)` then needed to accept any `setgroups()` and `setgid()` system calls. In order to avoid arbitrary programs from being able to leverage these calls independently, `mac_do(4)` would only authorize credentials transition requests from processes spawned from the `mdo(1)` program.

Because allowing any request from `setgroups()` and `setgid()` was a serious dent in reducing the impacts of an attack or a flaw, we modified `mac_do(4)` to validate the full credentials transition and its rules to say which groups can appear in the final credentials.

Validating or rejecting the full transition fundamentally requires atomicity, implying changes to the security API from traditional UNIX. A natural approach would be to add to the latter a transactional mode where successive calls amending credentials would not immediately apply changes but rather accumulate them for atomic application at final "commit". This approach would somehow facilitate amending existing programs as well as possible additions of credentials' attributes but was deemed relatively invasive in terms of kernel code and a paradigm change for the existing system calls' MAC hooks[22]. Instead, we settled for the alternative to have a new, separate system call, `setcred(2)`, that can set all credentials attributes at once. They are passed via a structure that can be extended or versioned through flags as needed. New MAC hooks are defined and are passed the current and requested credentials, allowing `mac_do(4)` to see the current and desired state at once and make decisions based on them.

Even after these changes, we have kept the restriction that `mac_do(4)` can only authorize processes spawned from the `mdo(1)` executable, as it may allow implementing additional transition restrictions in `mdo(1)` proper. A Google Summer of Code 2025 (GSoC 2025) student, Kushagra Srivastava, was tasked, among others, to bring configurability to this restriction, allowing an administrator to specify which executables `mac_do(4)` can authorize.

Some people may find strange, and even a potential security hazard, that code responsible for checking and deciding on credentials transitions based on rules is moved to the kernel instead of being executed in userland. While the ability to compromise the kernel would certainly be even more catastrophic than "setuid executables", we believe the former is much less likely to occur than the latter for the following reasons.

First, rules accepted by `mac_do(4)` are completely well-defined, self-contained, relatively simple to parse and hopefully to comprehend.

Second, "setuid executables" performing credentials changes generally involve a lot more components than `mac_do(4)` actually uses. The latter are essentially some parts of the MAC framework and the jail and OSD subsystems, which are pervasively used and tested and do not frequently or deeply change. The former are the libraries to read the password and group databases, which may involve network access, the userland configuration parser, and the code establishing all characteristics of the new session, including the credentials, which is sometimes part of a separate library, not even mentioning the usual userland support code, such as the dynamic linker.

"Setuid executables" performing credentials changes generally involve a lot more components than `mac_do(4)` actually uses.

Third, we have taken special care to design and write `mac_do(4)` in some of the cleanest and clearest ways, with special attention to understanding the constraints of the underlying subsystems and ensuring that the ones we rely on cannot be changed without our noticing via assertions. The result is that, despite copious testing, we have yet to find a bug in `mac_do(4)`'s core functionality (famous last words). Out of the few bug reports we received, only two turned out to be real problems in scenarios that admittedly were not well-considered nor tested initially[23], which led to performing another audit of the code. Our GSoC student was also tasked with developing automated tests, which should enter the official tree in the coming weeks. They will represent additional safeguards and will help maintain code quality as `mac_do(4)` and its dependent subsystems evolve.

What Lies Ahead

The essential message here is that, while we have a few simple short-term plans and more loose longer-term ones, future directions will depend for the most part on current or potential users' feedback. We are eager to hear suggestions for small improvements or entirely new features, whether you are already using `mac_do(4)`/`mdo(1)`, are planning to, or would like to but cannot because your use case is not covered by existing functionality. This will help us select what to work on while keeping the overall design sound. Even just saying you're using them is useful feedback, as it is good to know how many users we have and how they are using these tools.

In the short term, we expect to add auditing-like functionalities to `mac_do(4)`/`mdo(1)`. Displaying the final credentials passed to the kernel would help check if the invocation was correct with respect to the expected goals. Producing the target part of a `mac_do(4)`'s rule

authorizing exactly a specific `mdo(1)` call could help administrators build `mac_do(4)` configurations or better understand why some do not work as expected. Integration to the `audit(4)` subsystem would allow tracking credentials changes after the fact. Logging failed attempts through `syslog(3)` would match what `login(1)` and other credentials-changing program do. `mac_do(4)` will soon allow configuring the executables whose processes it will consider, with the aim to support thin-jails scenarios and other userland programs[24]. It should also monitor traditional system calls such as `setuid(2)` in addition to just `setcred(2)`, considering each call as a full transition on its own[24].

Longer term, we may consider providing `su`-like and `doas`-like functionalities, e.g., to ask for a password or perhaps more generally leveraging `pam(3)`, establish resource limits and other attributes as in a full login, or allow only certain commands to be launched. However, it is not yet clear how these functionalities could be fit into `mdo(1)`, as it is not a "setuid executable", and if different paths should be pursued instead.

Future directions will depend for the most part on current or potential users' feedback.

As an example, we have conducted a preliminary study on how to add support for requesting a password for certain credentials transitions. As `mdo(1)` can be launched by any user, we need a mechanism to check for a password against a password database which is not directly readable by everybody[25]. This situation is comparable to that of programs leveraging CAPSICUM's capability mode[26] which sometimes need to access data that require more privileges than they directly keep. That can be resolved by having an unrestricted process perform the necessary accesses on behalf of the process in capability mode. `libcasper(3)` is FreeBSD's implementation of that idea for a number of services, including `cap_pwd(3)` to access the password and group databases. Unfortunately, using `libcasper` as-is cannot work as `cap_enter()` creates and connects to a process launched with the same credentials. `mdo(1)` is going to need an outside daemon with privileges to provide the `cap_pwd(3)` service. We can also imagine a number of alternative approaches with varying development effort. They include pushing the password configuration entirely into `mac_do(4)` as for the rules, or turning `mdo(1)` into a "setuid" executable that however relinquishes root rights for most of its operation and crucially when calling `setcred(2)`, or instead leaving `mdo(1)` as it is and having a different "setuid" executable for these needs[27]. However, all of these alternatives except the first provide fewer security guarantees than the initial solution, and the first one is less flexible as it does not allow other forms of authentication nor additional transition restrictions that can be best imposed by userland[28].

We hope you will find `mac_do(4)`/`mdo(1)` useful! Please share your feedback and more generally other security needs you would like to see addressed, even if not necessarily directly connected to the framework presented here.

Footnotes

1. In reality, the special user ID 0. The name `root` resolves to ID 0, as may other names such as `toor`.
2. More precisely, the effective and saved user IDs, and the effective and saved group IDs, respectively. The saved user and group IDs are officially called the "Saved Set-User-ID" and "Saved Set-Group-ID" in the POSIX specification.
3. Other authentication mechanisms can be configured using PAM, see `pam(3)` for an introduction, `pam.conf(5)` for configuring particular applications, and `pam_unix(8)` for the canonical module.

4. As additional arguments to `su(1)` are passed to the target user's shell, the program and its arguments have to be passed through the shell's `-c` argument (or equivalent). For `sh(1)` and descendants, they must be grouped in a single argument that will be interpreted by the launched shell, sometimes requiring an additional level of quoting.
5. Mandatory Access Control. See `mac(4)`.
6. The updated `mdo(1)` described here will normally be shipped with FreeBSD 14.4.
7. There may be multiple users mapping to the same numerical ID. `doas(1)` has the flaw that it will silently consider the first matching user name. `mdo(1)` generally follows the conservative approach of not doing non-obvious operations silently, here not trying to use a matching user name, even if there is only one.
8. I.e., the real, effective and saved group IDs, by contrast with the supplementary groups.
9. To ease scripting, `-s` is actually compatible with `-G` and can be used to amend it, so it is in effect processed after `-G` even if it appears earlier on the command-line. Currently, though, using both `@` and `-G` is treated as an error (redundant specification), a limitation which may be lifted in the future.
10. The real user ID is matched, as it represents the user's identity, rather than the effective user ID, preventing by default another set of rules to apply for "setuid executables". That said, since unprivileged users are allowed to set the real user ID to the effective user ID on FreeBSD, this distinction is currently not an absolute restriction.
11. Since FreeBSD 15, a user's initial group from the password database is also installed as a supplementary group, which is also the case on Linux/glibc, NetBSD, OpenBSD, and illumos. For compatibility with FreeBSD 14.3, we demonstrate the target clause `+gid=80` here, which also works on 15.0, instead of `!gid=80`, which would allow the transition only on 15.0.
12. In more formal parlance, `gid` and `+gid` target clauses form a logical disjunction.
13. In more formal parlance, rules form a logical disjunction.
14. If `+gid=`. was replaced by `!gid=.`, the rule would allow a transition if and only if the current supplementary groups do not include 0, and not a transition to all current groups but 0. We may relax this constraint in the future.
15. There are some exceptions. We have seen one in the previous footnote. Another one is that, on one hand, the real, effective, and saved user IDs, and on the other hand, the real, effective, and saved group IDs are treated indifferently. Treating them separately was deemed to introduce additional complexity for meager benefit since FreeBSD's `setresuid()` currently allows an unprivileged process to set any of its user IDs to the value of any other one. We might want to disallow this behavior in the future.
16. Since FreeBSD 8.0.
17. Which always has a global jail ID of 0. Jail IDs are global, except that any process sees the ID of its immediately enclosing jail as 0.
18. In an earlier implementation, that parameter was called `mdo` and was intended to work like described here but did not due to bugs.
19. And thus, consistency issues.
20. Also known as Postel's Law. "Be liberal in what you accept, and conservative in what you send."
21. By the `/etc/rc.d/sysctl` script.
22. Either these hooks' existing implementations would need to start supporting the transactional mode, or we would bypass the hooks entirely, a change deemed too surprising to consumers.
23. Namely, using `mac_do(4)` when running with resource accounting functionality enabled, and running a 32-bit `mdo(1)` on a 64-bit architecture.
24. Most of the code for this functionality has been written during GSoC 2025 and should be integrated soon.
25. In order to avoid leaking password hashes that would allow offline attacks.
26. A process mode where most accesses to the global namespaces are restricted, and only existing file descriptors can be used.
27. That could take the form of first importing `doas(1)` into the base system and then tailoring it to our unique security features, although that would be a regression in terms of the granularity of target credentials. Alternatively, we could create an executable that would share part of its code and command-line interface with `mdo(1)`. Mixing both approaches to get the best of both worlds could also be viable.
28. But it has the benefit of not lessening the currently existing security guarantees, since the password would be checked by the kernel as well.
29. In the different real, effective and saved group IDs.

OLIVIER CERTNER has been continuously using FreeBSD on all his machines and those of some of the companies he worked with since the end of 2004. During this time, he has grown a set of private customizations including modifications to rc scripts and some kernel bits. After having worked for over 15 years in the CAD and finance sectors, he lately switched back to pure IT topics, and in particular operating system development. His main interests are centered around kernel development, with particular focuses on power management, security, scheduling, file systems and jails. He's currently a contractor for the FreeBSD Foundation.