Passkey + Simple Membership Integration Analysis

Can users authenticate with a passkey and retain access to Simple Membership protected content?

Executive summary

Yes — an integration can absolutely be forged, and it is small. Both plugins are well-designed for this exact scenario. The passkey plugin fires a custom action on every successful login, and Simple Membership already exposes a public bridge method (`SwpmAuth::login_to_swpm_using_wp_user()`) intended for "other methods of login authentication" — precisely the situation here.

Two paths are viable. The shortest is a one-line filter that tells Secure Passkeys to fire WordPress's native `wp_login` action — because Simple Membership is already listening on that hook and will bridge the session automatically. A slightly more defensive path hooks the plugin's own custom action and calls the SWPM bridge explicitly.

The only prerequisite is a data one: the WordPress user and the SWPM member must share the same email address, and the member account must not be in a `pending` / `activation_required` state.

Why protected content is hidden after a passkey login

Simple Membership does not rely on WordPress's auth cookie to gate content. It maintains its own parallel session, tracked by a separate cookie (simple_wp_membership_{COOKIEHASH} on HTTP, simple_wp_membership_sec_{COOKIEHASH} on HTTPS) and its own logged-in flag on the SwpmAuth singleton. Content protection runs inside the the_content filter, where SwpmAccessControl calls SwpmAuth::is_logged_in() — not the native WordPress equivalent. If that flag is false, the post body is swapped out for the "login required" message regardless of what WordPress thinks about the visitor.

SWPM's own login form takes username+password and calls SwpmAuth internally, so the parallel session is created as a side effect. Third-party login mechanisms — SSO, social login, API, and passkeys — need a different bridge. Simple Membership provides one: it listens on WordPress's wp_login action, and whenever that fires it calls login_to_swpm_using_wp_user() to mirror the WP login over into its own session.

Secure Passkeys, however, does not fire wp_login by default. After verifying the WebAuthn assertion it calls wp_set_current_user() and wp_set_auth_cookie() directly — WordPress is satisfied, the user is authenticated — but the wp_login action is gated behind a filter that ships with a default of false. The result: WordPress recognises the login, SWPM never hears about it, and the membership session is never created.

How Secure Passkeys signs the user in

The verification and login happen in a single action class. Its execute() method is the entire source of truth for what occurs after a valid passkey assertion.

File: `src/actions/secure-passkeys-web-authn-sign-in-action.php`

```
public function execute(int $user_id): bool
{
    $user = get_userdata($user_id);
    if (!$user) { return false; }

    $validate = apply_filters('secure_passkeys_web_authn_validate_user_sign_in', true,
$user);
    if (!$validate) { return false; }

    wp_set_current_user($user_id, $user->user_login);
    wp_set_auth_cookie($user_id, true);

    $enabled = apply_filters(
        'secure_passkeys_web_authn_validate_user_sign_in_enable_wp_login',
        false,            // <-- default: do NOT fire wp_login
        $user
    );
    if ($enabled) {
        do_action('wp_login', $user->user_login, $user);
    }

    do_action('secure_passkeys_web_authn_sign_in', $user);   // <-- always fires

    return true;
}
```

Three integration surfaces are relevant here:

- `secure_passkeys_web_authn_validate_user_sign_in_enable_wp_login` — a filter that, when returned as `true`, causes the plugin to fire WordPress's native `wp_login` action with the correct signature.
- `secure_passkeys_web_authn_sign_in` — a custom action that always fires after `wp_set_auth_cookie()`, passing the `WP_User` object. This is the cleanest, always-present extension point.
- `secure_passkeys_web_authn_validate_user_sign_in` — a pre-login filter that can veto the login (useful if a member's SWPM account is in a blocking state and you want to refuse the authentication entirely rather than let WP log them in and then have SWPM deny content).

How Simple Membership receives an external login

Simple Membership registers a listener on wp_login during plugin bootstrap. When any component — core WP, a social-login plugin, an SSO handler, the passkey plugin — fires that action, SWPM takes the WP_User object, looks up the SWPM member by email, and creates the parallel session.

File: `classes/class.simple-wp-membership.php  (line 88)`

```
add_action('wp_login', array(&$this, 'wp_login_hook_handler'), 10, 2);

public function wp_login_hook_handler($user_login, $user) {
    SwpmLog::log_auth_debug('wp_login hook triggered. Username: ' . $user_login, true);
    $auth = SwpmAuth::get_instance();
```

```
    if ($auth->is_logged_in()) { return; }
    $auth->login_to_swpm_using_wp_user($user);
}
```

The bridge method itself is the key public API. File: `classes/class.swpm-auth.php` (lines 405–424)

```
public function login_to_swpm_using_wp_user( $user ) {
    if ( $this->isLoggedIn ) { return false; }
    $email  = $user->user_email;
    $member = SwpmMemberUtils::get_user_by_email( $email );
    if ( empty( $member ) ) {
        // There is no swpm profile with this email.
        return false;
    }
    $this->userData   = $member;
    $this->isLoggedIn = true;

    $remember_me = isset($_REQUEST['rememberme']) && !empty($_REQUEST['rememberme'])
        ? $_REQUEST['rememberme'] : '';
    $this->set_cookie($remember_me);
    SwpmLog::log_auth_debug('Member has been logged in using WP User object.', true);
    $this->check_constraints();
    return true;
}
```

Three things to notice about this method. First, it explicitly identifies WP and SWPM users by email address — there is no foreign-key column linking the two tables. Second, it sets the SWPM cookie itself, so once it returns true the user's very next page load will see the membership-gated content. Third, a comment block further down in the same file confirms the design intent: "This function is NOT called for our plugin's standard login form submission. This is called for other methods of login authentication (for example, wp_authenticate_handler, auto_login, API addon etc)." Passkey login fits that pattern exactly.

The integration: two viable paths

Path A — Flip the filter (minimal change)

Because SWPM is already listening on wp_login, the simplest possible integration is to tell Secure Passkeys to fire it. A single filter return value is enough.

```
add_filter(
    'secure_passkeys_web_authn_validate_user_sign_in_enable_wp_login',
    '__return_true'
);
```

Effect: Secure Passkeys now fires do_action('wp_login', $user_login, $user) after wp_set_auth_cookie(). SWPM's wp_login_hook_handler receives it, calls login_to_swpm_using_wp_user($user), looks up the member by email, and writes the SWPM cookie. Protected content becomes visible on the very next request.

Pros: one line, uses only documented extension points on both sides, automatically benefits any other plugins that also listen to wp_login (analytics, audit logs, login trackers). Cons: it is a site-wide toggle — it enables wp_login for every passkey login, not just those of SWPM members. On a typical site where this is a feature, not a risk, Path A is the right answer.

Path B — Bridge explicitly (more defensive)

If the site owner would rather not fire wp_login globally from the passkey flow, the same result can be achieved by hooking Secure Passkeys's own custom action and calling the SWPM bridge directly. This is also the path to pick when there is non-trivial logic to run — for example, vetoing passkey logins for suspended SWPM accounts.

```
add_action( 'secure_passkeys_web_authn_sign_in', function ( $user ) {
    if ( ! class_exists( 'SwpmAuth' ) ) { return; }        // SWPM not loaded
    $swpm = SwpmAuth::get_instance();
    if ( $swpm->is_logged_in() ) { return; }               // already bridged
    $swpm->login_to_swpm_using_wp_user( $user );
}, 20, 1 );
```

Pros: narrower blast radius, no reliance on a global wp_login fan-out, easy place to add SWPM-specific guard logic (e.g., pre-check account_state and call SwpmAuth::logout() plus wp_logout() if the member is suspended). Cons: a few more lines, and — importantly — other plugins that listen on wp_login will still be bypassed by passkey logins unless the filter from Path A is also set.

Prerequisites and edge cases

The integration is as reliable as the data it rests on. The items below are worth verifying in advance — none are blockers, but each is a failure mode to be aware of.

- **Email linkage.** The WP user and SWPM member record are joined on email only. If a user has an SWPM membership under email A and a WP account under email B, `login_to_swpm_using_wp_user()` returns `false` silently. The passkey login still succeeds on the WP side; the member session simply never gets created. An audit script that compares the two tables before launch is a low-cost safety check.
- **Account state.** SWPM's `check_constraints()` runs after the cookie is set and can flip the session off again for members in `pending` or `activation_required` states. Expired accounts behave per the site's "enable expired account login" setting. This is identical to the behaviour of the shortcode login form — no special handling required — but a support team should be briefed that a passkey login with a hidden "activation required" response will look like "my passkey worked but content is still locked."
- **Load order.** SWPM registers its `wp_login` listener at plugin-bootstrap time, which is well before any AJAX request is handled. There is no race to worry about as long as both plugins are active.
- **"Remember me" semantics.** Secure Passkeys always calls `wp_set_auth_cookie($user_id, true)` — persistent on WordPress. SWPM's `login_to_swpm_using_wp_user()` reads `$_REQUEST['rememberme']`, which is not present in the passkey AJAX call — the SWPM cookie will therefore be a session cookie by default. This is a minor UX asymmetry. If persistent SWPM sessions are desired, the Path B handler can `$_REQUEST['rememberme'] = 1;` before calling the bridge.

- **Logout.** SWPM already hooks `wp_logout` and clears its cookie there. Nothing passkey-specific is needed on the logout path — WordPress logout will trigger SWPM logout as usual.
- **Registration.** This integration only bridges *existing* member accounts. If a WP user has registered a passkey but has never signed up for SWPM, the bridge will no-op. Whether that is a bug or a feature depends on the business rule — if every passkey user should automatically get a default membership, that has to be provisioned separately.

Recommendation

Implement Path A as the default: a single-line filter that enables wp_login in the Secure Passkeys flow. It is smaller, uses only the documented extension points, and benefits any other plugins on the site that depend on wp_login firing.

Prefer Path B if the site already has a reason to keep passkey logins from firing wp_login — for instance, if another plugin on wp_login would do something undesirable when invoked from the passkey flow, or if SWPM-specific guard logic (suspension checks, "remember me" overrides, audit logging) belongs in the integration.

Either way, wrap the integration in a tiny mu-plugin (or a small standalone plugin) rather than editing either vendor plugin. Both plugins expose stable public hooks for exactly this purpose, and upgrades on either side will not disturb the integration.

Appendix — hook and API reference

Summary of the surfaces relevant to this integration.

Name	Kind	Purpose
`secure_passkeys_web_authn_sign_in`	action	Always fires after successful passkey login. Receives WP_User. Best integration point.
`secure_passkeys_web_authn_validate_user_sign_in_enable_wp_login`	filter	Default false. Return true to make Secure Passkeys fire the native wp_login action.
`secure_passkeys_web_authn_validate_user_sign_in`	filter	Pre-auth veto. Return false to abort the passkey login before the cookie is set.
`wp_login`	action	WordPress core. SWPM's wp_login_hook_handler() listens here and calls the bridge.
`SwpmAuth::login_to_swpm_using_wp_user($user)`	method	Public bridge. Takes a WP_User, finds the

Name	Kind	Purpose
		SWPM member by email, sets the SWPM cookie.
`SwpmMemberUtils::get_user_by_email($email)`	method	Email lookup used internally; also useful for pre-flight existence checks.
`swpm_after_login_authentication`	action	Fires only from the SWPM shortcode login path — not reached by the bridge method. Do not rely on it for passkey flows.