Custom Upgrade Policies

Protecting the ability to upgrade a package on chain using a single key can pose a security risk for several reasons:

To address the security risk of single-key upgrade ownership poses while still providing the opportunity to upgrade live packages , Sui offers custom upgrade policies . These policies protect UpgradeCap access and issue UpgradeTicket objects that authorize upgrades on a case-by-case basis.

Sui comes with a set of built-in package compatibility policies, listed here from most to least strict:

Each of these policies, in the order listed, is a superset of the previous one in the type of changes allowed in the upgraded package.

When you publish a package, by default it adopts the most relaxed, compatible policy. You can publish a package as part of a transaction that can change the policy before the transaction successfully completes, making the package available on chain for the first time at the desired policy level, rather than at the default one.

You can change the current policy by calling one of the functions in sui::package (only_additive_upgrades , only_dep_upgrades , make_immutable) on the package's UpgradeCap and a policy can become only more restrictive. For example, after you call sui::package::only_dep_upgrades to restrict the policy to become additive, calling sui::package::only_additive_upgrades on the UpgradeCap of the same package results in an error.

Package upgrades must occur end-to-end in a single transaction and are composed of three commands:

While step 2 is a built-in command, steps 1 and 3 are implemented as Move functions. The Sui framework provides their most basic implementation:

These are the functions that sui client upgrade calls for authorization and commit. Custom upgrade policies work by guarding access to a package UpgradeCap (and therefore to calls of these functions) behind extra conditions that are specific to that policy (such as voting, governance, permission lists, timelocks, and so on).

Any pair of functions that produces an UpgradeTicket from an UpgradeCap and consumes an UpgradeReceipt to update an UpgradeCap constitutes a custom upgrade policy.

The UpgradeCap is the central type responsible for coordinating package upgrades.

Publishing a package creates the UpgradeCap object and upgrading the package updates that object. The owner of this object has permission to:

And its API guarantees the following properties:

An UpgradeTicket is proof that an upgrade has been authorized. This authorization is specific to:

When you attempt to run the upgrade, the validator checks that the upgrade it is about to perform matches the upgrade that was authorized along all those lines, and does not perform the upgrade if any of these criteria are not met.

After creating an UpgradeTicket , you must use it within that transaction (you cannot store it for later, drop it, or burn it), or the transaction fails.

The UpgradeTicket digest field comes from the digest parameter to authorize_upgrade , which the caller must supply. While authorize_upgrade does not process the digest , custom policies can use it to authorize only upgrades that it has seen the bytecode or source code for ahead of time. Sui calculates the digest as follows:

Refer to the implementation for digest calculation for more information, but in most cases, you can rely on the Move toolchain to output the digest as part of the build, when passing the --dump-bytecode-as-base64 flag:

The UpgradeReceipt is proof that the Upgrade command ran successfully, and Sui added the new package to the set of created objects for the transaction. It is used to update its UpgradeCap (identified by cap: ID) with the ID of the latest package in its family (package: ID).

After Sui creates an UpgradeReceipt , you must use it to update its UpgradeCap within the same transaction (you cannot store it for later, drop it, or burn it), or the transaction fails.

When writing custom upgrade policies, prefer:

These best practices help uphold informed user consent and bounded risk by making it clear what a package's upgrade policy is at the moment a user locks value into it, and ensuring that the policy does not evolve to be more permissive with time, without the package user realizing and choosing to accept the new terms.

Time to put everything into practice by writing a toy upgrade policy that only authorizes upgrades on a particular day of the week (of the package creator's choosing).

Start by creating a new Move package for the upgrade policy:

The command creates a policy directory with a sources folder and Move.toml manifest.

In the sources folder, create a source file named day_of_week.move . Copy and paste the following code into the file:

This code includes a constructor and defines the object type for the custom upgrade policy.

You then need to add a function to authorize an upgrade, if on the correct day of the week. First, define a couple of constants, one for the error code that identifies an attempted upgrade on a day the policy doesn't allow, and another to define the number of milliseconds in a day (to be used shortly). Add these definitions directly under the current ENotWeekDay one.

After the new_policy function, add a week_day function to get the current weekday. As promised, the function uses the MS_IN_DAY constant you defined earlier.

This function uses the epoch timestamp from TxContext rather than Clock because it needs only daily granularity, which means the upgrade transactions don't require consensus.

Next, add an authorize_upgrade function that calls the previous function to get the current day of the week, then checks whether that value violates the policy, returning the ENotAllowedDay error value if it does.

The signature of a custom authorize_upgrade can be different from the signature of sui::package::authorize_upgrade as long as it returns an UpgradeTicket .

Finally, provide implementations of commit_upgrade and make_immutable that delegate to their respective functions in sui::package :

The final code in your day_of_week.move file should resemble the following:

Use the sui client publish command to publish the policy.

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

Console output

A successful publish returns the following:

Following best practices, use the Sui Client CLI to call sui::package::make_immutable on the UpgradeCap to make the policy immutable. In your shell, create a variable upgradecap and set its value to the UpgradeCap object ID listed in the Object Changes section of your publish response. Of course, the object ID for your upgrade capability is different than the following example.

Console output

A successful call returns a response similar to the following:

With a policy now available on chain, you need a package to upgrade. This topic creates a basic package and references it in the following scenarios, but you can use any package you might have available instead of creating a new one.

If you don't have a package available, use the sui move new command to create the template for a new package called example .

In the example/sources directory, create an example.move file with the following code:

The instruction that follows publishes this example package and then upgrades it to change the value in the Event it emits. Because you are using a custom upgrade policy, you need to use the TypeScript SDK to build the package's publish and upgrade commands.

Create a new directory to store a Node.js project. You can use the npm init function to create the package.json , or manually create the file. Depending on your approach to creating package.json , populate or add the following JSON to it:

Open a terminal or console to the root of your Node.js project. Run the following command to add the Sui TypeScript SDK as a dependency:

In the root of your Node.js project, create a script file named publish.js . Open the file for editing and define some constants:

Next, add boilerplate code to get the signer key pair for the currently active address in the Sui Client CLI:

Next, define the path of the package you are publishing. The following snippet assumes that the package is in a sibling directory to publish.js , called example :

Next, build the package:

Next, construct the transaction to publish the package. Wrap its UpgradeCap in a "day of the week" policy, which permits upgrades on Tuesdays, and send the new policy back:

And finally, execute that transaction and display its effects to the console. The following snippet assumes that you're running your examples against a local network. Pass devnet , testnet , or mainnet to the getFullnodeUrl() function to run on Devnet, Testnet, or Mainnet respectively:

publish.js

Save your publish.js file, and then use Node.js to run the script:

Console output

If the script is successful, the console prints the following response:

If you receive a ReferenceError: fetch is not defined error, use Node.js version 18 or greater.

Use the CLI to test that your newly published package works:

Console output

A successful call responds with the following:

If you used the example package provided, notice you have an Events section that contains a field x with value 41 .

With your package published, you can prepare an upgrade.js script to perform an upgrade using the new policy. It behaves identically to publish.js up until building the package. When building the package, the script also captures its digest , and the transaction now performs the three upgrade commands (authorize, execute, commit). The full script for upgrade.js follows:

If today is not Tuesday, wait until next Tuesday to run the script, when your policy allows you to perform upgrades. At that point, update your example.move so the event is emitted with a different constant and use Node.js to run the upgrade script:

Console output

If the script is successful (and today is Tuesday), your console displays the following response:

Use the Sui Client CLI to test the upgraded package (the package ID is different from the original version of your example package):

Console output

If successful, the console prints the following response:

Now, the Events section emitted for the x field has a value of 42 (changed from the original 41).

If you attempt the first upgrade before Tuesday or you change the constant again and try the upgrade the following day, the script receives a response that includes an error similar to the following, which indicates that the upgrade aborted with code 2 (ENotAllowedDay):

Compatibility

Sui comes with a set of built-in package compatibility policies, listed here from most to least strict:

Each of these policies, in the order listed, is a superset of the previous one in the type of changes allowed in the upgraded package.

When you publish a package, by default it adopts the most relaxed, compatible policy. You can publish a package as part of a transaction that can change the policy before the transaction successfully completes, making the package available on chain for the first time at the desired policy level, rather than at the default one.

You can change the current policy by calling one of the functions in sui::package (only_additive_upgrades , only_dep_upgrades , make_immutable) on the package's UpgradeCap and a policy can become only more restrictive. For example, after you call sui::package::only_dep_upgrades to restrict the policy to become additive, calling sui::package::only_additive_upgrades on the UpgradeCap of the same package results in an error.

Package upgrades must occur end-to-end in a single transaction and are composed of three commands:

While step 2 is a built-in command, steps 1 and 3 are implemented as Move functions. The Sui framework provides their most basic implementation:

These are the functions that sui client upgrade calls for authorization and commit. Custom upgrade policies work by guarding access to a package UpgradeCap (and therefore to calls of these functions) behind extra conditions that are specific to that policy (such as voting, governance, permission lists, timelocks, and so on).

Any pair of functions that produces an UpgradeTicket from an UpgradeCap and consumes an UpgradeReceipt to update an UpgradeCap constitutes a custom upgrade policy.

The UpgradeCap is the central type responsible for coordinating package upgrades.

Publishing a package creates the UpgradeCap object and upgrading the package updates that object. The owner of this object has permission to:

And its API guarantees the following properties:

An UpgradeTicket is proof that an upgrade has been authorized. This authorization is specific to:

When you attempt to run the upgrade, the validator checks that the upgrade it is about to perform matches the upgrade that was authorized along all those lines, and does not perform the upgrade if any of these criteria are not met.

After creating an UpgradeTicket , you must use it within that transaction (you cannot store it for later, drop it, or burn it), or the transaction fails.

The UpgradeTicket digest field comes from the digest parameter to authorize_upgrade , which the caller must supply. While authorize_upgrade does not process the digest , custom policies can use it to authorize only upgrades that it has seen the bytecode or source code for ahead of time. Sui calculates the digest as follows:

Refer to the implementation for digest calculation for more information, but in most cases, you can rely on the Move toolchain to output the digest as part of the build, when passing the --dump-bytecode-as-base64 flag:

The UpgradeReceipt is proof that the Upgrade command ran successfully, and Sui added the new package to the set of created objects for the transaction. It is used to update its UpgradeCap (identified by cap: ID) with the ID of the latest package in its family (package: ID).

After Sui creates an UpgradeReceipt , you must use it to update its UpgradeCap within the same transaction (you cannot store it for later, drop it, or burn it), or the transaction fails.

When writing custom upgrade policies, prefer:

These best practices help uphold informed user consent and bounded risk by making it clear what a package's upgrade policy is at the moment a user locks value into it, and ensuring that the policy does not evolve to be more permissive with time, without the package user realizing and choosing to accept the new terms.

Time to put everything into practice by writing a toy upgrade policy that only authorizes upgrades on a particular day of the week (of the package creator's choosing).

Start by creating a new Move package for the upgrade policy:

The command creates a policy directory with a sources folder and Move.toml manifest.

In the sources folder, create a source file named day_of_week.move . Copy and paste the following code into the file:

This code includes a constructor and defines the object type for the custom upgrade policy.

You then need to add a function to authorize an upgrade, if on the correct day of the week. First, define a couple of constants, one for the error code that identifies an attempted upgrade on a day the policy doesn't allow, and another to define the number of milliseconds in a day (to be used shortly). Add these definitions directly under the current ENotWeekDay one.

After the new_policy function, add a week_day function to get the current weekday. As promised, the function uses the MS_IN_DAY constant you defined earlier.

This function uses the epoch timestamp from TxContext rather than Clock because it needs only daily granularity, which means the upgrade transactions don't require consensus.

Next, add an authorize_upgrade function that calls the previous function to get the current day of the week, then checks whether that value violates the policy, returning the ENotAllowedDay error value if it does.

The signature of a custom authorize_upgrade can be different from the signature of sui::package::authorize_upgrade as long as it returns an UpgradeTicket .

Finally, provide implementations of commit_upgrade and make_immutable that delegate to their respective functions in sui::package :

The final code in your day_of_week.move file should resemble the following:

Use the sui client publish command to publish the policy.

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

Console output

A successful publish returns the following:

Following best practices, use the Sui Client CLI to call sui::package::make_immutable on the UpgradeCap to make the policy immutable. In your shell, create a variable upgradecap and set its value to the UpgradeCap object ID listed in the Object Changes section of your publish response. Of course, the object ID for your upgrade capability is different than the following example.

Console output

A successful call returns a response similar to the following:

With a policy now available on chain, you need a package to upgrade. This topic creates a basic package and references it in the following scenarios, but you can use any package you might have available instead of creating a new one.

If you don't have a package available, use the sui move new command to create the template for a new package called example .

In the example/sources directory, create an example.move file with the following code:

The instruction that follows publishes this example package and then upgrades it to change the value in the Event it emits. Because you are using a custom upgrade policy, you need to use the TypeScript SDK to build the package's publish and upgrade commands.

Create a new directory to store a Node.js project. You can use the npm init function to create the package.json , or manually create the file. Depending on your approach to creating package.json , populate or add the following JSON to it:

Open a terminal or console to the root of your Node.js project. Run the following command to add the Sui TypeScript SDK as a dependency:

In the root of your Node.js project, create a script file named publish.js . Open the file for editing and define some constants:

Next, add boilerplate code to get the signer key pair for the currently active address in the Sui Client CLI:

Next, define the path of the package you are publishing. The following snippet assumes that the package is in a sibling directory to publish.js , called example :

Next, build the package:

Next, construct the transaction to publish the package. Wrap its UpgradeCap in a "day of the week" policy, which permits upgrades on Tuesdays, and send the new policy back:

And finally, execute that transaction and display its effects to the console. The following snippet assumes that you're running your examples against a local network. Pass devnet , testnet , or mainnet to the getFullnodeUrl() function to run on Devnet, Testnet, or

Mainnet respectively:

publish.js

Save your publish.js file, and then use Node.js to run the script:

Console output

If the script is successful, the console prints the following response:

If you receive a ReferenceError: fetch is not defined error, use Node.js version 18 or greater.

Use the CLI to test that your newly published package works:

Console output

A successful call responds with the following:

If you used the example package provided, notice you have an Events section that contains a field x with value 41 .

With your package published, you can prepare an upgrade.js script to perform an upgrade using the new policy. It behaves identically to publish.js up until building the package. When building the package, the script also captures its digest , and the transaction now performs the three upgrade commands (authorize, execute, commit). The full script for upgrade.js follows:

If today is not Tuesday, wait until next Tuesday to run the script, when your policy allows you to perform upgrades. At that point, update your example.move so the event is emitted with a different constant and use Node.js to run the upgrade script:

Console output

If the script is successful (and today is Tuesday), your console displays the following response:

Use the Sui Client CLI to test the upgraded package (the package ID is different from the original version of your example package):

Console output

If successful, the console prints the following response:

Now, the Events section emitted for the x field has a value of 42 (changed from the original 41).

If you attempt the first upgrade before Tuesday or you change the constant again and try the upgrade the following day, the script receives a response that includes an error similar to the following, which indicates that the upgrade aborted with code 2 (ENotAllowedDay):

Upgrade overview

Package upgrades must occur end-to-end in a single transaction and are composed of three commands:

While step 2 is a built-in command, steps 1 and 3 are implemented as Move functions. The Sui framework provides their most basic implementation:

These are the functions that sui client upgrade calls for authorization and commit. Custom upgrade policies work by guarding access to a package UpgradeCap (and therefore to calls of these functions) behind extra conditions that are specific to that policy (such as voting, governance, permission lists, timelocks, and so on).

Any pair of functions that produces an UpgradeTicket from an UpgradeCap and consumes an UpgradeReceipt to update an UpgradeCap constitutes a custom upgrade policy.

The UpgradeCap is the central type responsible for coordinating package upgrades.

Publishing a package creates the UpgradeCap object and upgrading the package updates that object. The owner of this object has permission to:

And its API guarantees the following properties:

An UpgradeTicket is proof that an upgrade has been authorized. This authorization is specific to:

When you attempt to run the upgrade, the validator checks that the upgrade it is about to perform matches the upgrade that was authorized along all those lines, and does not perform the upgrade if any of these criteria are not met.

After creating an UpgradeTicket , you must use it within that transaction (you cannot store it for later, drop it, or burn it), or the transaction fails.

The UpgradeTicket digest field comes from the digest parameter to authorize_upgrade , which the caller must supply. While authorize_upgrade does not process the digest , custom policies can use it to authorize only upgrades that it has seen the bytecode or source code for ahead of time. Sui calculates the digest as follows:

Refer to the implementation for digest calculation for more information, but in most cases, you can rely on the Move toolchain to output the digest as part of the build, when passing the --dump-bytecode-as-base64 flag:

The UpgradeReceipt is proof that the Upgrade command ran successfully, and Sui added the new package to the set of created objects for the transaction. It is used to update its UpgradeCap (identified by cap: ID) with the ID of the latest package in its family (package: ID).

After Sui creates an UpgradeReceipt , you must use it to update its UpgradeCap within the same transaction (you cannot store it for later, drop it, or burn it), or the transaction fails.

When writing custom upgrade policies, prefer:

These best practices help uphold informed user consent and bounded risk by making it clear what a package's upgrade policy is at the moment a user locks value into it, and ensuring that the policy does not evolve to be more permissive with time, without the package user realizing and choosing to accept the new terms.

Time to put everything into practice by writing a toy upgrade policy that only authorizes upgrades on a particular day of the week (of the package creator's choosing).

Start by creating a new Move package for the upgrade policy:

The command creates a policy directory with a sources folder and Move.toml manifest.

In the sources folder, create a source file named day_of_week.move . Copy and paste the following code into the file:

This code includes a constructor and defines the object type for the custom upgrade policy.

You then need to add a function to authorize an upgrade, if on the correct day of the week. First, define a couple of constants, one for the error code that identifies an attempted upgrade on a day the policy doesn't allow, and another to define the number of milliseconds in a day (to be used shortly). Add these definitions directly under the current ENotWeekDay one.

After the new_policy function, add a week_day function to get the current weekday. As promised, the function uses the MS_IN_DAY constant you defined earlier.

This function uses the epoch timestamp from TxContext rather than Clock because it needs only daily granularity, which means the upgrade transactions don't require consensus.

Next, add an authorize_upgrade function that calls the previous function to get the current day of the week, then checks whether that value violates the policy, returning the ENotAllowedDay error value if it does.

The signature of a custom authorize_upgrade can be different from the signature of sui::package::authorize_upgrade as long as it returns an UpgradeTicket .

Finally, provide implementations of commit_upgrade and make_immutable that delegate to their respective functions in sui::package :

The final code in your day_of_week.move file should resemble the following:

Use the sui client publish command to publish the policy.

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

Console output

A successful publish returns the following:

Following best practices, use the Sui Client CLI to call sui::package::make_immutable on the UpgradeCap to make the policy immutable. In your shell, create a variable upgradecap and set its value to the UpgradeCap object ID listed in the Object Changes section of your publish response. Of course, the object ID for your upgrade capability is different than the following example.

Console output

A successful call returns a response similar to the following:

With a policy now available on chain, you need a package to upgrade. This topic creates a basic package and references it in the following scenarios, but you can use any package you might have available instead of creating a new one.

If you don't have a package available, use the sui move new command to create the template for a new package called example .

In the example/sources directory, create an example.move file with the following code:

The instruction that follows publishes this example package and then upgrades it to change the value in the Event it emits. Because you are using a custom upgrade policy, you need to use the TypeScript SDK to build the package's publish and upgrade commands.

Create a new directory to store a Node.js project. You can use the npm init function to create the package.json , or manually create the file. Depending on your approach to creating package.json , populate or add the following JSON to it:

Open a terminal or console to the root of your Node.js project. Run the following command to add the Sui TypeScript SDK as a dependency:

In the root of your Node.js project, create a script file named publish.js . Open the file for editing and define some constants:

Next, add boilerplate code to get the signer key pair for the currently active address in the Sui Client CLI:

Next, define the path of the package you are publishing. The following snippet assumes that the package is in a sibling directory to publish.js , called example :

Next, build the package:

Next, construct the transaction to publish the package. Wrap its UpgradeCap in a "day of the week" policy, which permits upgrades on Tuesdays, and send the new policy back:

And finally, execute that transaction and display its effects to the console. The following snippet assumes that you're running your examples against a local network. Pass devnet , testnet , or mainnet to the getFullnodeUrl() function to run on Devnet, Testnet, or Mainnet respectively:

publish.js

Save your publish.js file, and then use Node.js to run the script:

Console output

If the script is successful, the console prints the following response:

If you receive a ReferenceError: fetch is not defined error, use Node.js version 18 or greater.

Use the CLI to test that your newly published package works:

Console output

A successful call responds with the following:

If you used the example package provided, notice you have an Events section that contains a field x with value 41 .

With your package published, you can prepare an upgrade.js script to perform an upgrade using the new policy. It behaves identically to publish.js up until building the package. When building the package, the script also captures its digest , and the transaction now performs the three upgrade commands (authorize, execute, commit). The full script for upgrade.js follows:

If today is not Tuesday, wait until next Tuesday to run the script, when your policy allows you to perform upgrades. At that point, update your example.move so the event is emitted with a different constant and use Node.js to run the upgrade script:

Console output

If the script is successful (and today is Tuesday), your console displays the following response:

Use the Sui Client CLI to test the upgraded package (the package ID is different from the original version of your example package):

Console output

If successful, the console prints the following response:

Now, the Events section emitted for the x field has a value of 42 (changed from the original 41).

If you attempt the first upgrade before Tuesday or you change the constant again and try the upgrade the following day, the script receives a response that includes an error similar to the following, which indicates that the upgrade aborted with code 2 (ENotAllowedDay):

UpgradeCap

The UpgradeCap is the central type responsible for coordinating package upgrades.

Publishing a package creates the UpgradeCap object and upgrading the package updates that object. The owner of this object has permission to:

And its API guarantees the following properties:

An UpgradeTicket is proof that an upgrade has been authorized. This authorization is specific to:

When you attempt to run the upgrade, the validator checks that the upgrade it is about to perform matches the upgrade that was authorized along all those lines, and does not perform the upgrade if any of these criteria are not met.

After creating an UpgradeTicket , you must use it within that transaction (you cannot store it for later, drop it, or burn it), or the transaction fails.

The UpgradeTicket digest field comes from the digest parameter to authorize_upgrade , which the caller must supply. While authorize_upgrade does not process the digest , custom policies can use it to authorize only upgrades that it has seen the bytecode or source code for ahead of time. Sui calculates the digest as follows:

Refer to the implementation for digest calculation for more information, but in most cases, you can rely on the Move toolchain to output the digest as part of the build, when passing the --dump-bytecode-as-base64 flag:

The UpgradeReceipt is proof that the Upgrade command ran successfully, and Sui added the new package to the set of created objects for the transaction. It is used to update its UpgradeCap (identified by cap: ID) with the ID of the latest package in its family (package: ID).

After Sui creates an UpgradeReceipt , you must use it to update its UpgradeCap within the same transaction (you cannot store it for later, drop it, or burn it), or the transaction fails.

When writing custom upgrade policies, prefer:

These best practices help uphold informed user consent and bounded risk by making it clear what a package's upgrade policy is at the moment a user locks value into it, and ensuring that the policy does not evolve to be more permissive with time, without the package user realizing and choosing to accept the new terms.

Time to put everything into practice by writing a toy upgrade policy that only authorizes upgrades on a particular day of the week (of the package creator's choosing).

Start by creating a new Move package for the upgrade policy:

The command creates a policy directory with a sources folder and Move.toml manifest.

In the sources folder, create a source file named day_of_week.move . Copy and paste the following code into the file:

This code includes a constructor and defines the object type for the custom upgrade policy.

You then need to add a function to authorize an upgrade, if on the correct day of the week. First, define a couple of constants, one for the error code that identifies an attempted upgrade on a day the policy doesn't allow, and another to define the number of milliseconds in a day (to be used shortly). Add these definitions directly under the current ENotWeekDay one.

After the new_policy function, add a week_day function to get the current weekday. As promised, the function uses the MS_IN_DAY constant you defined earlier.

This function uses the epoch timestamp from TxContext rather than Clock because it needs only daily granularity, which means the upgrade transactions don't require consensus.

Next, add an authorize_upgrade function that calls the previous function to get the current day of the week, then checks whether that value violates the policy, returning the ENotAllowedDay error value if it does.

The signature of a custom authorize_upgrade can be different from the signature of sui::package::authorize_upgrade as long as it returns an UpgradeTicket .

Finally, provide implementations of commit_upgrade and make_immutable that delegate to their respective functions in sui::package :

The final code in your day_of_week.move file should resemble the following:

Use the sui client publish command to publish the policy.

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

Console output

A successful publish returns the following:

Following best practices, use the Sui Client CLI to call sui::package::make_immutable on the UpgradeCap to make the policy immutable. In your shell, create a variable upgradecap and set its value to the UpgradeCap object ID listed in the Object Changes section of your publish response. Of course, the object ID for your upgrade capability is different than the following example.

Console output

A successful call returns a response similar to the following:

With a policy now available on chain, you need a package to upgrade. This topic creates a basic package and references it in the following scenarios, but you can use any package you might have available instead of creating a new one.

If you don't have a package available, use the sui move new command to create the template for a new package called example .

In the example/sources directory, create an example.move file with the following code:

The instruction that follows publishes this example package and then upgrades it to change the value in the Event it emits. Because you are using a custom upgrade policy, you need to use the TypeScript SDK to build the package's publish and upgrade commands.

Create a new directory to store a Node.js project. You can use the npm init function to create the package.json , or manually create the file. Depending on your approach to creating package.json , populate or add the following JSON to it:

Open a terminal or console to the root of your Node.js project. Run the following command to add the Sui TypeScript SDK as a dependency:

In the root of your Node.js project, create a script file named publish.js . Open the file for editing and define some constants:

Next, add boilerplate code to get the signer key pair for the currently active address in the Sui Client CLI:

Next, define the path of the package you are publishing. The following snippet assumes that the package is in a sibling directory to publish.js , called example :

Next, build the package:

Next, construct the transaction to publish the package. Wrap its UpgradeCap in a "day of the week" policy, which permits upgrades on Tuesdays, and send the new policy back:

And finally, execute that transaction and display its effects to the console. The following snippet assumes that you're running your examples against a local network. Pass devnet , testnet , or mainnet to the getFullnodeUrl() function to run on Devnet, Testnet, or

Mainnet respectively:

publish.js

Save your publish.js file, and then use Node.js to run the script:

Console output

If the script is successful, the console prints the following response:

If you receive a ReferenceError: fetch is not defined error, use Node.js version 18 or greater.

Use the CLI to test that your newly published package works:

Console output

A successful call responds with the following:

If you used the example package provided, notice you have an Events section that contains a field x with value 41 .

With your package published, you can prepare an upgrade.js script to perform an upgrade using the new policy. It behaves identically to publish.js up until building the package. When building the package, the script also captures its digest , and the transaction now performs the three upgrade commands (authorize, execute, commit). The full script for upgrade.js follows:

If today is not Tuesday, wait until next Tuesday to run the script, when your policy allows you to perform upgrades. At that point, update your example.move so the event is emitted with a different constant and use Node.js to run the upgrade script:

Console output

If the script is successful (and today is Tuesday), your console displays the following response:

Use the Sui Client CLI to test the upgraded package (the package ID is different from the original version of your example package):

Console output

If successful, the console prints the following response:

Now, the Events section emitted for the x field has a value of 42 (changed from the original 41).

If you attempt the first upgrade before Tuesday or you change the constant again and try the upgrade the following day, the script receives a response that includes an error similar to the following, which indicates that the upgrade aborted with code 2 (ENotAllowedDay):

UpgradeTicket

An UpgradeTicket is proof that an upgrade has been authorized. This authorization is specific to:

When you attempt to run the upgrade, the validator checks that the upgrade it is about to perform matches the upgrade that was authorized along all those lines, and does not perform the upgrade if any of these criteria are not met.

After creating an UpgradeTicket , you must use it within that transaction (you cannot store it for later, drop it, or burn it), or the transaction fails.

The UpgradeTicket digest field comes from the digest parameter to authorize_upgrade , which the caller must supply. While authorize_upgrade does not process the digest , custom policies can use it to authorize only upgrades that it has seen the bytecode or source code for ahead of time. Sui calculates the digest as follows:

Refer to the implementation for digest calculation for more information, but in most cases, you can rely on the Move toolchain to output the digest as part of the build, when passing the --dump-bytecode-as-base64 flag:

The UpgradeReceipt is proof that the Upgrade command ran successfully, and Sui added the new package to the set of created objects for the transaction. It is used to update its UpgradeCap (identified by cap: ID) with the ID of the latest package in its family (package: ID).

After Sui creates an UpgradeReceipt , you must use it to update its UpgradeCap within the same transaction (you cannot store it for later, drop it, or burn it), or the transaction fails.

When writing custom upgrade policies, prefer:

These best practices help uphold informed user consent and bounded risk by making it clear what a package's upgrade policy is at the moment a user locks value into it, and ensuring that the policy does not evolve to be more permissive with time, without the package user realizing and choosing to accept the new terms.

Time to put everything into practice by writing a toy upgrade policy that only authorizes upgrades on a particular day of the week (of the package creator's choosing).

Start by creating a new Move package for the upgrade policy:

The command creates a policy directory with a sources folder and Move.toml manifest.

In the sources folder, create a source file named day_of_week.move . Copy and paste the following code into the file:

This code includes a constructor and defines the object type for the custom upgrade policy.

You then need to add a function to authorize an upgrade, if on the correct day of the week. First, define a couple of constants, one for the error code that identifies an attempted upgrade on a day the policy doesn't allow, and another to define the number of milliseconds in a day (to be used shortly). Add these definitions directly under the current ENotWeekDay one.

After the new_policy function, add a week_day function to get the current weekday. As promised, the function uses the MS_IN_DAY constant you defined earlier.

This function uses the epoch timestamp from TxContext rather than Clock because it needs only daily granularity, which means the upgrade transactions don't require consensus.

Next, add an authorize_upgrade function that calls the previous function to get the current day of the week, then checks whether that value violates the policy, returning the ENotAllowedDay error value if it does.

The signature of a custom authorize_upgrade can be different from the signature of sui::package::authorize_upgrade as long as it returns an UpgradeTicket .

Finally, provide implementations of commit_upgrade and make_immutable that delegate to their respective functions in sui::package :

The final code in your day_of_week.move file should resemble the following:

Use the sui client publish command to publish the policy.

Beginning with the Sui v1.24.1 [release](#) , the --gas-budget option is no longer required for CLI commands.

Console output

A successful publish returns the following:

Following best practices, use the Sui Client CLI to call sui::package::make_immutable on the UpgradeCap to make the policy immutable. In your shell, create a variable upgradecap and set its value to the UpgradeCap object ID listed in the Object Changes section of your publish response. Of course, the object ID for your upgrade capability is different than the following example.

Console output

A successful call returns a response similar to the following:

With a policy now available on chain, you need a package to upgrade. This topic creates a basic package and references it in the following scenarios, but you can use any package you might have available instead of creating a new one.

If you don't have a package available, use the sui move new command to create the template for a new package called example .

In the example/sources directory, create an example.move file with the following code:

The instruction that follows publishes this example package and then upgrades it to change the value in the Event it emits. Because you are using a custom upgrade policy, you need to use the TypeScript SDK to build the package's publish and upgrade commands.

Create a new directory to store a Node.js project. You can use the npm init function to create the package.json , or manually create the file. Depending on your approach to creating package.json , populate or add the following JSON to it:

Open a terminal or console to the root of your Node.js project. Run the following command to add the Sui TypeScript SDK as a dependency:

In the root of your Node.js project, create a script file named publish.js . Open the file for editing and define some constants:

Next, add boilerplate code to get the signer key pair for the currently active address in the Sui Client CLI:

Next, define the path of the package you are publishing. The following snippet assumes that the package is in a sibling directory to publish.js , called example :

Next, build the package:

Next, construct the transaction to publish the package. Wrap its UpgradeCap in a "day of the week" policy, which permits upgrades on Tuesdays, and send the new policy back:

And finally, execute that transaction and display its effects to the console. The following snippet assumes that you're running your examples against a local network. Pass devnet , testnet , or mainnet to the getFullnodeUrl() function to run on Devnet, Testnet, or Mainnet respectively:

publish.js

Save your publish.js file, and then use Node.js to run the script:

Console output

If the script is successful, the console prints the following response:

If you receive a ReferenceError: fetch is not defined error, use Node.js version 18 or greater.

Use the CLI to test that your newly published package works:

Console output

A successful call responds with the following:

If you used the example package provided, notice you have an Events section that contains a field x with value 41 .

With your package published, you can prepare an upgrade.js script to perform an upgrade using the new policy. It behaves identically to publish.js up until building the package. When building the package, the script also captures its digest , and the transaction now performs the three upgrade commands (authorize, execute, commit). The full script for upgrade.js follows:

If today is not Tuesday, wait until next Tuesday to run the script, when your policy allows you to perform upgrades. At that point, update your example.move so the event is emitted with a different constant and use Node.js to run the upgrade script:

Console output

If the script is successful (and today is Tuesday), your console displays the following response:

Use the Sui Client CLI to test the upgraded package (the package ID is different from the original version of your example package):

Console output

If successful, the console prints the following response:

Now, the Events section emitted for the x field has a value of 42 (changed from the original 41).

If you attempt the first upgrade before Tuesday or you change the constant again and try the upgrade the following day, the script receives a response that includes an error similar to the following, which indicates that the upgrade aborted with code 2 (ENotAllowedDay):

UpgradeReceipt

The UpgradeReceipt is proof that the Upgrade command ran successfully, and Sui added the new package to the set of created objects for the transaction. It is used to update its UpgradeCap (identified by cap: ID) with the ID of the latest package in its family (package: ID).

After Sui creates an UpgradeReceipt , you must use it to update its UpgradeCap within the same transaction (you cannot store it for later, drop it, or burn it), or the transaction fails.

When writing custom upgrade policies, prefer:

These best practices help uphold informed user consent and bounded risk by making it clear what a package's upgrade policy is at the moment a user locks value into it, and ensuring that the policy does not evolve to be more permissive with time, without the package user realizing and choosing to accept the new terms.

Time to put everything into practice by writing a toy upgrade policy that only authorizes upgrades on a particular day of the week (of the package creator's choosing).

Start by creating a new Move package for the upgrade policy:

The command creates a policy directory with a sources folder and Move.toml manifest.

In the sources folder, create a source file named day_of_week.move . Copy and paste the following code into the file:

This code includes a constructor and defines the object type for the custom upgrade policy.

You then need to add a function to authorize an upgrade, if on the correct day of the week. First, define a couple of constants, one for the error code that identifies an attempted upgrade on a day the policy doesn't allow, and another to define the number of milliseconds in a day (to be used shortly). Add these definitions directly under the current ENotWeekDay one.

After the new_policy function, add a week_day function to get the current weekday. As promised, the function uses the MS_IN_DAY constant you defined earlier.

This function uses the epoch timestamp from TxContext rather than Clock because it needs only daily granularity, which means the upgrade transactions don't require consensus.

Next, add an authorize_upgrade function that calls the previous function to get the current day of the week, then checks whether that value violates the policy, returning the ENotAllowedDay error value if it does.

The signature of a custom authorize_upgrade can be different from the signature of sui::package::authorize_upgrade as long as it returns an UpgradeTicket .

Finally, provide implementations of commit_upgrade and make_immutable that delegate to their respective functions in sui::package :

The final code in your day_of_week.move file should resemble the following:

Use the sui client publish command to publish the policy.

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

Console output

A successful publish returns the following:

Following best practices, use the Sui Client CLI to call sui::package::make_immutable on the UpgradeCap to make the policy immutable. In your shell, create a variable upgradecap and set its value to the UpgradeCap object ID listed in the Object Changes section of your publish response. Of course, the object ID for your upgrade capability is different than the following example.

Console output

A successful call returns a response similar to the following:

With a policy now available on chain, you need a package to upgrade. This topic creates a basic package and references it in the following scenarios, but you can use any package you might have available instead of creating a new one.

If you don't have a package available, use the sui move new command to create the template for a new package called example .

In the example/sources directory, create an example.move file with the following code:

The instruction that follows publishes this example package and then upgrades it to change the value in the Event it emits. Because you are using a custom upgrade policy, you need to use the TypeScript SDK to build the package's publish and upgrade commands.

Create a new directory to store a Node.js project. You can use the npm init function to create the package.json , or manually create the file. Depending on your approach to creating package.json , populate or add the following JSON to it:

Open a terminal or console to the root of your Node.js project. Run the following command to add the Sui TypeScript SDK as a dependency:

In the root of your Node.js project, create a script file named publish.js . Open the file for editing and define some constants:

Next, add boilerplate code to get the signer key pair for the currently active address in the Sui Client CLI:

Next, define the path of the package you are publishing. The following snippet assumes that the package is in a sibling directory to publish.js , called example :

Next, build the package:

Next, construct the transaction to publish the package. Wrap its UpgradeCap in a "day of the week" policy, which permits upgrades on Tuesdays, and send the new policy back:

And finally, execute that transaction and display its effects to the console. The following snippet assumes that you're running your examples against a local network. Pass devnet , testnet , or mainnet to the getFullnodeUrl() function to run on Devnet, Testnet, or Mainnet respectively:

publish.js

Save your publish.js file, and then use Node.js to run the script:

Console output

If the script is successful, the console prints the following response:

If you receive a ReferenceError: fetch is not defined error, use Node.js version 18 or greater.

Use the CLI to test that your newly published package works:

Console output

A successful call responds with the following:

If you used the example package provided, notice you have an Events section that contains a field x with value 41 .

With your package published, you can prepare an upgrade.js script to perform an upgrade using the new policy. It behaves identically to publish.js up until building the package. When building the package, the script also captures its digest , and the transaction now performs the three upgrade commands (authorize, execute, commit). The full script for upgrade.js follows:

If today is not Tuesday, wait until next Tuesday to run the script, when your policy allows you to perform upgrades. At that point, update your example.move so the event is emitted with a different constant and use Node.js to run the upgrade script:

Console output

If the script is successful (and today is Tuesday), your console displays the following response:

Use the Sui Client CLI to test the upgraded package (the package ID is different from the original version of your example package):

Console output

If successful, the console prints the following response:

Now, the Events section emitted for the x field has a value of 42 (changed from the original 41).

If you attempt the first upgrade before Tuesday or you change the constant again and try the upgrade the following day, the script receives a response that includes an error similar to the following, which indicates that the upgrade aborted with code 2 (ENotAllowedDay):

Isolating policies

When writing custom upgrade policies, prefer:

These best practices help uphold informed user consent and bounded risk by making it clear what a package's upgrade policy is at the moment a user locks value into it, and ensuring that the policy does not evolve to be more permissive with time, without the package user realizing and choosing to accept the new terms.

Time to put everything into practice by writing a toy upgrade policy that only authorizes upgrades on a particular day of the week (of the package creator's choosing).

Start by creating a new Move package for the upgrade policy:

The command creates a policy directory with a sources folder and Move.toml manifest.

In the sources folder, create a source file named day_of_week.move . Copy and paste the following code into the file:

This code includes a constructor and defines the object type for the custom upgrade policy.

You then need to add a function to authorize an upgrade, if on the correct day of the week. First, define a couple of constants, one for the error code that identifies an attempted upgrade on a day the policy doesn't allow, and another to define the number of milliseconds in a day (to be used shortly). Add these definitions directly under the current ENotWeekDay one.

After the new_policy function, add a week_day function to get the current weekday. As promised, the function uses the MS_IN_DAY constant you defined earlier.

This function uses the epoch timestamp from TxContext rather than Clock because it needs only daily granularity, which means the upgrade transactions don't require consensus.

Next, add an authorize_upgrade function that calls the previous function to get the current day of the week, then checks whether that value violates the policy, returning the ENotAllowedDay error value if it does.

The signature of a custom authorize_upgrade can be different from the signature of sui::package::authorize_upgrade as long as it returns an UpgradeTicket .

Finally, provide implementations of commit_upgrade and make_immutable that delegate to their respective functions in sui::package :

The final code in your day_of_week.move file should resemble the following:

Use the sui client publish command to publish the policy.

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

Console output

A successful publish returns the following:

Following best practices, use the Sui Client CLI to call sui::package::make_immutable on the UpgradeCap to make the policy immutable. In your shell, create a variable upgradecap and set its value to the UpgradeCap object ID listed in the Object Changes section of your publish response. Of course, the object ID for your upgrade capability is different than the following example.

Console output

A successful call returns a response similar to the following:

With a policy now available on chain, you need a package to upgrade. This topic creates a basic package and references it in the following scenarios, but you can use any package you might have available instead of creating a new one.

If you don't have a package available, use the sui move new command to create the template for a new package called example .

In the example/sources directory, create an example.move file with the following code:

The instruction that follows publishes this example package and then upgrades it to change the value in the Event it emits. Because you are using a custom upgrade policy, you need to use the TypeScript SDK to build the package's publish and upgrade commands.

Create a new directory to store a Node.js project. You can use the npm init function to create the package.json , or manually create the file. Depending on your approach to creating package.json , populate or add the following JSON to it:

Open a terminal or console to the root of your Node.js project. Run the following command to add the Sui TypeScript SDK as a dependency:

In the root of your Node.js project, create a script file named publish.js . Open the file for editing and define some constants:

Next, add boilerplate code to get the signer key pair for the currently active address in the Sui Client CLI:

Next, define the path of the package you are publishing. The following snippet assumes that the package is in a sibling directory to publish.js , called example :

Next, build the package:

Next, construct the transaction to publish the package. Wrap its UpgradeCap in a "day of the week" policy, which permits upgrades on Tuesdays, and send the new policy back:

And finally, execute that transaction and display its effects to the console. The following snippet assumes that you're running your examples against a local network. Pass devnet , testnet , or mainnet to the getFullnodeUrl() function to run on Devnet, Testnet, or Mainnet respectively:

publish.js

Save your publish.js file, and then use Node.js to run the script:

Console output

If the script is successful, the console prints the following response:

If you receive a ReferenceError: fetch is not defined error, use Node.js version 18 or greater.

Use the CLI to test that your newly published package works:

Console output

A successful call responds with the following:

If you used the example package provided, notice you have an Events section that contains a field x with value 41 .

With your package published, you can prepare an upgrade.js script to perform an upgrade using the new policy. It behaves identically to publish.js up until building the package. When building the package, the script also captures its digest , and the transaction now performs the three upgrade commands (authorize, execute, commit). The full script for upgrade.js follows:

If today is not Tuesday, wait until next Tuesday to run the script, when your policy allows you to perform upgrades. At that point, update your example.move so the event is emitted with a different constant and use Node.js to run the upgrade script:

Console output

If the script is successful (and today is Tuesday), your console displays the following response:

Use the Sui Client CLI to test the upgraded package (the package ID is different from the original version of your example package):

Console output

If successful, the console prints the following response:

Now, the Events section emitted for the x field has a value of 42 (changed from the original 41).

If you attempt the first upgrade before Tuesday or you change the constant again and try the upgrade the following day, the script receives a response that includes an error similar to the following, which indicates that the upgrade aborted with code 2 (ENotAllowedDay):

Example: "Day of the Week" upgrade policy

Time to put everything into practice by writing a toy upgrade policy that only authorizes upgrades on a particular day of the week (of the package creator's choosing).

Start by creating a new Move package for the upgrade policy:

The command creates a policy directory with a sources folder and Move.toml manifest.

In the sources folder, create a source file named day_of_week.move . Copy and paste the following code into the file:

This code includes a constructor and defines the object type for the custom upgrade policy.

You then need to add a function to authorize an upgrade, if on the correct day of the week. First, define a couple of constants, one for the error code that identifies an attempted upgrade on a day the policy doesn't allow, and another to define the number of milliseconds in a day (to be used shortly). Add these definitions directly under the current ENotWeekDay one.

After the new_policy function, add a week_day function to get the current weekday. As promised, the function uses the MS_IN_DAY constant you defined earlier.

This function uses the epoch timestamp from TxContext rather than Clock because it needs only daily granularity, which means the upgrade transactions don't require consensus.

Next, add an authorize_upgrade function that calls the previous function to get the current day of the week, then checks whether that value violates the policy, returning the ENotAllowedDay error value if it does.

The signature of a custom authorize_upgrade can be different from the signature of sui::package::authorize_upgrade as long as it returns an UpgradeTicket .

Finally, provide implementations of commit_upgrade and make_immutable that delegate to their respective functions in sui::package :

The final code in your day_of_week.move file should resemble the following:

Use the sui client publish command to publish the policy.

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

Console output

A successful publish returns the following:

Following best practices, use the Sui Client CLI to call sui::package::make_immutable on the UpgradeCap to make the policy immutable. In your shell, create a variable upgradecap and set its value to the UpgradeCap object ID listed in the Object Changes section of your publish response. Of course, the object ID for your upgrade capability is different than the following example.

Console output

A successful call returns a response similar to the following:

With a policy now available on chain, you need a package to upgrade. This topic creates a basic package and references it in the following scenarios, but you can use any package you might have available instead of creating a new one.

If you don't have a package available, use the sui move new command to create the template for a new package called example .

In the example/sources directory, create an example.move file with the following code:

The instruction that follows publishes this example package and then upgrades it to change the value in the Event it emits. Because you are using a custom upgrade policy, you need to use the TypeScript SDK to build the package's publish and upgrade commands.

Create a new directory to store a Node.js project. You can use the npm init function to create the package.json , or manually create the file. Depending on your approach to creating package.json , populate or add the following JSON to it:

Open a terminal or console to the root of your Node.js project. Run the following command to add the Sui TypeScript SDK as a dependency:

In the root of your Node.js project, create a script file named publish.js . Open the file for editing and define some constants:

Next, add boilerplate code to get the signer key pair for the currently active address in the Sui Client CLI:

Next, define the path of the package you are publishing. The following snippet assumes that the package is in a sibling directory to publish.js , called example :

Next, build the package:

Next, construct the transaction to publish the package. Wrap its UpgradeCap in a "day of the week" policy, which permits upgrades on Tuesdays, and send the new policy back:

And finally, execute that transaction and display its effects to the console. The following snippet assumes that you're running your examples against a local network. Pass devnet , testnet , or mainnet to the getFullnodeUrl() function to run on Devnet, Testnet, or Mainnet respectively:

publish.js

Save your publish.js file, and then use Node.js to run the script:

Console output

If the script is successful, the console prints the following response:

If you receive a ReferenceError: fetch is not defined error, use Node.js version 18 or greater.

Use the CLI to test that your newly published package works:

Console output

A successful call responds with the following:

If you used the example package provided, notice you have an Events section that contains a field x with value 41 .

With your package published, you can prepare an upgrade.js script to perform an upgrade using the new policy. It behaves identically to publish.js up until building the package. When building the package, the script also captures its digest , and the transaction now performs the three upgrade commands (authorize, execute, commit). The full script for upgrade.js follows:

If today is not Tuesday, wait until next Tuesday to run the script, when your policy allows you to perform upgrades. At that point, update your example.move so the event is emitted with a different constant and use Node.js to run the upgrade script:

Console output

If the script is successful (and today is Tuesday), your console displays the following response:

Use the Sui Client CLI to test the upgraded package (the package ID is different from the original version of your example package):

Console output

If successful, the console prints the following response:

Now, the Events section emitted for the x field has a value of 42 (changed from the original 41).

If you attempt the first upgrade before Tuesday or you change the constant again and try the upgrade the following day, the script receives a response that includes an error similar to the following, which indicates that the upgrade aborted with code 2 (ENotAllowedDay):